*Appeared in Proceedings DSD'08 (Parma, Italy, September 2008), p. 242–249* November 26, 2008

# Mapping a Fault-Tolerant Distributed Algorithm to Systems on Chip

Gottfried Fuchs, Matthias Függer, Ulrich Schmid, Andreas Steininger
*Vienna University of Technology, Embedded Computing Systems Group*
{fuchs, fuegger, s, steininger}@ecs.tuwien.ac.at

## Abstract

*Systems on chip (SoC) have much in common with traditional (networked) distributed systems in that they consist of largely independent components with dedicated communication interfaces. Therefore the adoption of classic distributed algorithms for SoCs suggests itself. The implementation complexity of these algorithms, however, significantly depends on the underlying failure models. In traditional software-based solutions this is normally not an issue, such that the most unconstrained, namely the Byzantine, failure model is often applied here. Our case study of a hardware-implemented tick synchronization algorithm shows, however, that in an SoC-implementation substantial hardware savings can result from restricting the failure model to benign failures (omissions, crashes). On the downside, it turns out that such restricted failure models have a fairly poor coverage with respect to the hardware faults occurring in practice, and that additional measures to enforce these restrictions may entail an implementation overhead that outweighs the gain obtained in the implementation of a simpler algorithm. As a remedy we investigate the potential of failure transformation in this context and show that this technique may indeed yield an optimized overall solution.*

## 1. Introduction

VLSI technology trends like continuously shrinking feature sizes allow for ever increasing complexity of digital circuits — millions of transistors are nowadays packed on a single die [12]. These advances also changed the internal structure and operation of digital circuits, however: First, the enormous complexity of current and future VLSI designs makes a design partitioning approach mandatory. Moreover, clock frequencies in the range of several GHz imply that signal transitions cannot traverse the whole chip within a single clock cycle. And finally, the decreasing voltage swing — required for achieving high clock frequencies with reasonable power dissipation — diminishes the circuits' robustness and amplifies the adverse effects of $\alpha$-particle and neutron hits [15], as well as crosstalk sensitivity [16, 17]. Consequently, a modern VLSI circuit can no longer be viewed as a monolithic block of hardware that operates synchronously throughout the whole chip, but has to be viewed as a complex system of interacting functional units, namely, a SoC (System on Chip).

Given the resulting increase of the transient fault rate in such devices, faults have to be considered part of normal operation, rather than exceptional operating conditions as in traditional VLSI designs. Consequently, fault-tolerance mechanisms will likely be vital for future VLSI circuits [2, 5]. Since there is a wide spectrum of fault-tolerance mechanisms available, ranging from the transistor level up to the system level, choosing the most appropriate technique will become an issue.

Interestingly, a closer look at SoCs reveals striking similarities with networked distributed systems: Dealing with asynchrony, erroneous components and unreliable links has been the topic of research in the fault-tolerant distributed algorithms community for several decades. A wealth of results regarding failure models, algorithms and protocols, as well as a huge body of theoretical findings related to the solvability of certain problems and lower bounds on the achievable performance have hence been established in the past. This includes problems and algorithms for adding fault tolerance to networked systems, which can be achieved either by making use of knowledge specific to the application domain or, in a systematic way, independently of the application. Clearly the second approach is more promising because of its generality. Systematic fault-tolerance approaches can be classified into those which provide a fault-tolerant time base and those which offer a fault-tolerant state to the application. The first can be achieved by fault-tolerant clock synchronization algorithms, while the second can be obtained by consensus and distributed shared memory algorithms. As the latter algorithms, e.g. consensus, are hard to implement without a fault-tolerant time base, in this paper, we concentrate on how to map a well-known fault-tolerant clock synchronization scheme to SoCs.

Actually, some proofs of concept for the applicability of distributed algorithms in SoCs have already appeared in literature. For example, in [8, 9], it has been shown that self-stabilizing algorithms [7], which recover even from massive transient faults, can be employed in VLSI chips. Similarly, in our DARTS[1] project (Distributed Algorithms for Robust Tick-Synchronization), we have implemented a fault-tolerant clocking scheme for SoCs by adapting a simple clock synchronization algorithm to the requirements and constraints of asynchronous digital logic [11].

## 2. From software-based algorithms to hardware

Adapting fault-tolerant distributed algorithms to the new application domain of SoCs is non-trivial for several reasons: A major concern is the inherent massive parallelism implied by concurrently operating logic gates, which does not very well match the standard assumption of a (small) set of concurrent processing elements. Furthermore, operations that may seem simple in software based distributed algorithms (such as multiplication, ordering or even comparison of integers) are far too costly in terms of area and/or speed to be *directly* implemented in HW. Another important issue where standard distributed computing assumptions must be adapted to match the requirements of SoCs are failure models. A failure model is a set of assumptions that specifies how and how many components in a distributed system may fail without jeopardizing its correct operation. For example, popular failure models assume that $f$ out of the $n$ processors in a distributed system may just stop operating (crash failures) or may behave totally arbitrary (Byzantine failures) [14]. Such failure models are of course very different from the fault models commonly used in VLSI research, which consider effects like stuck-at faults or delay faults at signal level. The failure mode assumptions sometimes defined for hardware blocks (such as fail silent, e.g.) would definitely be the better match here. Unfortunately, however, they can usually only be defined for relatively large, independent functional units, which limits their applicability for low-level issues.

**Distributed System Models:** When analyzing the effect of mapping failure models from distributed systems to SoCs it is important to unambiguously define the system model considered. Typically a distributed system is modeled as a finite set of processes (nodes) $V$ and a (directed or undirected) communication graph $G = (V, E)$, $E \subseteq V \times V$, where $(v, w) \in E$ means that node $v$ can produce content that can be read by node $w$. A distributed algorithm is an assignment of an algorithm (state machine) to each of the nodes in $V$. The system model defines how algorithms are executed and how they communicate, and specifies the allowed failure patterns.

Throughout this paper we assume a set of $n$ fully connected nodes (modeling the SoC components), communicating via FIFO message passing links. The communication delay $\delta$, i.e., the time between sending and receiving a message $m$, is arbitrary except for a bounded ratio $\Theta$ of the delays for sending the same $m$ to different receivers [13, 24]. Assuming that a node constitutes a fault containment region, an appropriate failure model essentially boils down to a set of constraints on how correct nodes perceive the messages sent by faulty nodes.

From the VLSI point of view identifying the weakest distributed failure model corresponding to a given VLSI fault model is a non-trivial task.

## 3. Towards a fault-tolerant time base: DARTS

As already mentioned in Section 1, future generation chips will be more susceptible to transient faults, including upsets in combinational logic. This threat also exists for the clock distribution network — a very prominent single point of failure in most current VLSI designs. As a consequence, fault-tolerant clock generation and clock distribution schemes will likely attract increasing attention. Unfortunately, traditional voter-based fault-tolerance schemes [4, 18, 21, 22] cannot deal with the substantial skew usually present in high speed designs. The GALS approach (Globally Asynchronous Locally Synchronous) [3], on the other hand, assumes multiple independent — and hence not mutually synchronized — clock sources. GALS thus sacrifices a substantial advantage of synchronous systems, namely, the global time base.

In our DARTS project, we are addressing the above mentioned problems of traditional fault-tolerant clocking as well as the GALS approach by exploiting the similarity between traditional distributed systems and SoCs. The structure of a system based on the DARTS clocking scheme is shown in Figure 1. An SoC using the DARTS clocking scheme is naturally partitioned into several functional units $Fu_n$, each of which is equipped with a dedicated *tick generation unit* (TG-Alg) that supplies a local clock signal to the respective $Fu$ (or groups of $Fu$'s). All TG-Algs interact over a dedicated *tick generation network* (TG-Net), thereby not only keeping their local clock signals in approximate synchrony, but also effectively *generating* the clock by virtue of the mutual feedback. Consequently, no crystal oscillator is required in the DARTS clocking approach.

In DARTS, we chose (a variant of) the consistent broadcast primitive by Srikanth and Toueg [19] for the TG-Alg implementation. The algorithm is used as a building block in several distributed algorithms, including fault-tolerant clock synchronization in distributed systems. The key problem solved in DARTS is the implementation of the — originally software based — algorithm in asynchronous digital

[1]The project DARTS project received funding from the Austrian bm:vit (FIT-IT, contract no. 809456-SCK/SAI).

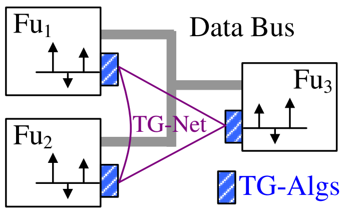

**Figure 1. FT distributed tick generation.**

logic. Apart from substantial adaptations of the original Algorithm 1, this also required refined techniques for proving correctness and analyzing the achievable synchronization precision and clock frequency [11].

Algorithm 1 relies on a fully connected network, atomicity of actions, and the exchange of messages that carry an unbounded integer identifier. In order to facilitate a hardware implementation, essential adaptations had to be applied to this algorithm: First, since one cannot assume the existence of a clock for implementing the TG-Algs, they had to be built using asynchronous logic. As there are no natural atomic operations (sequential "computing steps" as assumed when writing algorithms in software) in the execution of such devices, we had to enforce them. This can be achieved by two methods: (i) explicit handshaking and (ii) implicit handshaking by timing constraints. Technique (i) is what is typically used in asynchronous hardware design. The intended fault tolerance of our TG-Algs is in clear contrast to explicit handshaking between all components, however, as a faulty one could cause the whole system to fail as well. Technique (ii) ensures that concurrent computation of sequential events does not change the event order of these processing results. In our TG-Algs the timing constraints, responsible for the correct event order, can only be ensured by employing independent circuitry for processing rising and falling clock signal transitions — the so called interlocking of rising and falling transitions.

Moreover, to keep the TG-Net as light-weight as possible, only single signal rails could reasonably be used here. Together with the fact maximum speed is desireable for

**Algorithm 1** Byzantine tolerant tick generation [23]

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

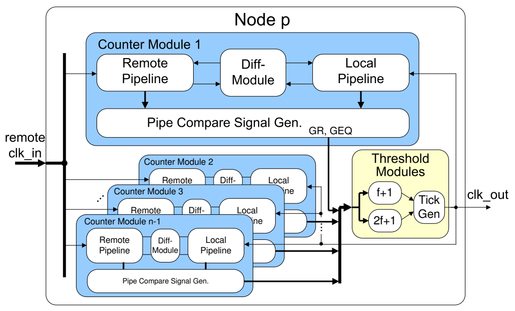

**Figure 2. The DARTS implementation.**

clock generation, this forced us to substitute the algorithm's tick($k$) messages by single anonymous signal transitions. As a result, no information except event occurrences can be conveyed via the TG-Net, and the tick numbers required by the algorithm must be maintained at every receiver. The resulting TG-Alg hardware design is shown in Figure 2.

To maintain the required difference of local ($k$) and remote ($\ell$) clock ticks, we combined two *Elastic Pipelines* (EP) [20] per remote TG-Alg with a custom *Difference-Module* (Diff-Module, DM) in between. While the EPs serve as FIFO buffers for clock signal transitions, the DM is responsible for removing transitions that match in both pipelines. Note that we proved in [11] that $\ell - k$ remains bounded, with a bound that depends on the ratio $\Theta$ of the slowest vs. fastest delay of certain critical paths in the system only. Consequently, the pipeline depth and hence the whole design of the TG units is quite independent of the implementation technology. The circuitry named *Pipe Compare Signal Generator* (PCSG) represents simple glue logic, responsible for evaluating whether or not $\ell - k \geq 0$ and/or $\ell - k \geq 1$. This status information is finally fed into *Threshold Modules* (THM), which implement the relay- and increment-rules of Algorithm 1 and generate a new clock tick as soon as enough ($f+1$ resp. $2f+1$) matching ticks have been received from remote TG-Algs. The parallel execution of two rules in Algorithm 1 represents a substantial problem for the hardware implementation: In order to enforce the mutual exclusion of the "firing" of the rules (as implied by the "[once]" in the algorithm), we had to build separate function blocks for rising and falling clock edges and establish an appropriate interlocking scheme.

As proved in [11], the DARTS clocking scheme guarantees a worst-case synchronization precision $\pi$ in the range of a few clock ticks; the actual value of $\pi$ again depends on the ratio $\Theta$. To tolerate up to $f$ Byzantine faulty TG-Algs, a system of $n \geq 3f+2$ TG-Alg nodes is required[2]. Our theoretical results have been confirmed by an FPGA proto-

[2]Note that the lower bound of $n \geq 3f+1$ nodes is sacrificed intentionally here to get a considerably better precision $\pi$.

type implementation [10], which also provided the required proof of concept for the ASIC implementation currently being tested and evaluated.

## 4. Impact of the failure model

We have chosen the Byzantine failure model for the implementation of our DARTS TG-Algs, since there is no better choice w.r.t. failures: With a faulty system component being allowed to behave arbitrarily, the issue of failure model coverage does not arise. However, clock synchronization resilient to $f$ Byzantine failures requires $n \geq 3f+1$ and a $2f+1$-connected (i.e., essentially fully connected) point-to-point network [6]. The optimal coverage of the chosen Byzantine failure model hence comes at a high price in terms of implementation costs (gate count):

**Connectivity:** The almost fully connected point-to-point network implies a quadratic growth of the number of links with $n$ and thus with $f$, i.e., is in $\mathcal{O}(f^2)$. The design complexity for properly routing the TG-Net without violating any constraint is manageable, however, since the routing is relatively uncritical.

**Elastic pipelines:** For every clock input from a remote TG-Alg, each node requires one pair of local/remote EPs, yielding a $\mathcal{O}(n(n-1)) = \mathcal{O}(f^2)$ area requirement system-wide.

**Rules & THMs:** Algorithm 1 employs two rules ($f+1$ relay and $2f+1$ increment) and hence requires four THMs (two thresholds plus the above mentioned separation of logic for rising and falling edges in order to attain the interlocking). A standard CMOS implementation of the THMs exhibits a very unfavorable scaling with $n$: If $A_{THM}(k,m)$ denotes the area complexity of a single $k$-of-$m$ THM, it turned out that $A_{THM}(k,m) \propto \binom{k}{m}$ in this case. The circuit of Fig. 2 employs both $f+1$-of-$3f+1$ and $2f+1$-of-$3f+1$ THMs. Both have the same area requirement, since a $2f+1$-of-$3f+1$ THM can be obtained by negating all inputs and the output of a $f+1$-of-$3f+1$ THM. The total area requirement per TG-Alg is hence $4A_{THM}(f+1, 3f+1) \propto 4\binom{3f+1}{f+1}$. As an example let us assume that we want to tolerate 3 instead of 2 Byzantine faulty TG-Algs. We thus need 4-of-10 THMs instead of 3-of-7 THMs, which leads to an area increase by a factor of $A_{THM}(4,10)/A_{THM}(3,7) \approx 11$.

Notice, however, that the situation would be radically different if an alternative target technology were used for THMs, like threshold logic [1]. Here, $A_{THM}(k,m)$ is linear in $m, k$ and hence scales well with $f$ and $n$ — the area increase $A_{THM}(4,10)/A_{THM}(3,7)$ would hence probably be negligible here.

Further notice that the THM, in spite of its seemingly exotic nature, constitutes an essential building block not only for our specific example but for the implementation of fault-tolerant distributed algorithms in general.

Anyway, depending on the implementation technology, significant savings in terms of chip area can be expected by relaxing the failure model: Reducing the type (= adverse power) of the allowed failures may result in (a) a lower number $n$ of TG-Algs needed to tolerate a fixed number $f$ of failures, (b) less connectivity requirements, and (c) fewer rules — obtained at the price of reduced hardware fault coverage. Such considerations are not limited to our DARTS clocking scheme; similar issues should arise for almost any fault-tolerant distributed algorithm adapted to SoCs.

## 5. The gain of a weaker failure model

The question arises whether the implementation effort can be decreased significantly by considering more restricted failure models and whether these failure models are appropriate with respect to coverage of hardware faults. We will hence present a number of algorithms for tick synchronization in the presence of weaker failure semantics.

In the following we will consider failures where it holds that: If $p$ receives message $m$ from $q$, then $p$ can be sure that $q$ sent message $m$ and that it arrived timely. We start with the simplest failure semantics, namely, clean crashes.

**Clean crash.** A node $p$ is said to have cleanly-crashed at time $t$, if *(i)* all messages sent by $p$ before time $t$ are received by all other nodes and *(ii)* all messages sent by $p$ at time $t$ or later are not received by any node.

An algorithm solving the tick generation problem for clean crashes together with a proof was presented in [23] and is depicted in Alg. 2. To tolerate $f$ cleanly-crashed nodes, $n \geq f+2$ nodes are required. The algorithm has only one rule, which promises a significant simplification of the hardware implementation, since the parallelism has been eliminated at least at the algorithm level.

**Crash:** A node $p \in V$ is said to have crashed at time $t$, if *(i)* all messages sent by $p$ before time $t$ are received by all other nodes, *(ii)* a message sent by $p$ at time $t$ is received by nodes $V' \subseteq V$ ($V' = V$ in case of clean crashes) and *(iii)* a message sent by $p$ later than time $t$ is not received by any node.

Interestingly, Algorithm 2 not only tolerates clean-crashes, but also allows up to $f$ nodes to crash in an unclean

**Algorithm 2** Crash tolerant tick generation

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

**Algorithm 3** Omission tolerant tick generation

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

way if $n \geq f+2$. However, the precision $\pi$ becomes dependent on $f$ when using this algorithm with unclean crashes.

The situation changes completely if we extend the crash failure semantics to omission failures.

**Omission:** A node $p$ is said to have an omission failure, if a message sent by $p$ is received by a subset of $V$.

In contrast to crashed nodes, omissive nodes are allowed to act asymmetrically an unbounded number of times, thus rendering Algorithm 2 useless. Algorithm 3 was taken from [23], where it was presented as an algorithm tolerating (unclean) crashes. It turns out that this algorithm even allows to tolerate up to $f$ omissive nodes, if $n \geq 2f + 2$. The possibility to tolerate an unbounded number of asymmetrically received messages, however, comes at the price of a significant increase in the hardware implementation effort (more nodes required, two rules). Note that Algorithm 3 even provides a precision independent of $f$ in presence of up to $f$ omissive nodes. Since a crashed node can always be modeled as an omissive node, this algorithm can hence also be used to guarantee a precision independent of $f$ in case of (unclean) crashes.

To allow a comparison w.r.t. the hardware implementation effort, we built Algorithms 1–3 in $0.18\mu$ SEU hardened CMOS technology for $f = 3$ and obtained the area estimates presented in Table 1.

## 6. The coverage of a weaker failure model

Viewed from a less formal and more hardware-related point, a *clean crash* can be translated into two requirements in our setting: *(i) Causality:* A fault must not produce any extra transitions, or move them ahead of time — every transition experienced by a receiver must be the result of the algorithm's proper operation. *(ii) Symmetry:* The perception of a fault must be the same for all receivers. This is not the case if its perception depends on a receiver's specific properties like physical location, threshold or speed.

Requirement (i) is clearly easily violated, e.g., by transients on a communication link that produce extra glitches. Notice that even a stuck-at fault may create an "early" transition and thus invalidates (i). Threats for requirement (ii) are marginal effects like short glitches or undefined voltage levels created by a fault. An open defect in a communication link may not only produce both of these, but will definitely cause asymmetric perception if it affects only a branch of the network.

These examples make it evident that the assumption of clean crashes is very optimistic in a typical hardware setting, since it rules out many commonly encountered fault scenarios.

Unfortunately, relaxing the assumptions to *crash failures* does not help much: Requirement (i) remains unchanged, while with respect to (ii) asymmetric perception is allowed now once before the faulty input channel turns mute.

The assumption of *omission failures* clearly disburdens us of requirement (ii), but it is still hard to argue that even (i) alone can be maintained in a realistic setting. So, in summary, the adoption of a failure model more restricted than the Byzantine model can generally not be justified for a hardware implementation in practice. However, as we have seen in Section 5, there is a potential for significant savings by relaxing failure semantics, so it might be worthwhile to invest some efforts on enforcing the requirements. In particular, there is the concept of "simulation" and "failure transformation" in the distributed systems community, see e.g. [19], that may come for a rescue here.

## 7. Failure transformation as a remedy

Assume that up to $f$ failures of type $A$ (e.g. Byzantine) might occur in a given distributed system. Then there exist failure transformation algorithms, which provide services (e.g. broadcast) to an upper layer that are proved to fail in a more restricted way $B$ (e.g. by clean-crashes only).

Unfortunately generic failure transformation algorithms for turning Byzantine failures into weaker failures typically need to communicate multiple times between node $p$ and $q$ to provide a simulation of a single message from $p$ to $q$ at the upper layer. In case of clock/tick synchronization, a multi-round transformation is definitely not acceptable from a performance point of view. However, there exists a trans-

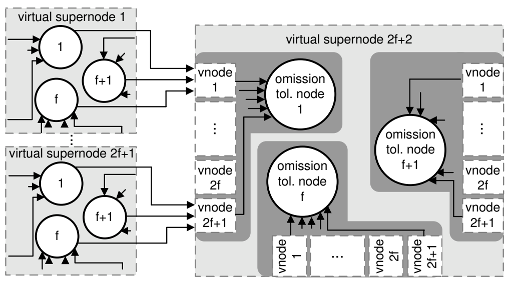

**Figure 3. Simulation of an omission tolerant algorithm.**

**Table 1. Byzantine, crash and omission tolerant algorithms hardware effort of a single node.**

| | f=3 | | | fault-tolerant for | | |
|---|---|---|---|---|---|---|
| | Area in [$\mu m^2$] | # of nodes | # of links | byzantines | ommissions | crashes |
| Algorithm 1 | 483859 | 11 | 11x10 | x | x | x |
| Algorithm 2 | 9810 | 5 | 5x4 | - | - | x |
| Algorithm 3 | 33188 | 8 | 8x7 | - | x | x |

formation which turns $f$ Byzantine failures at a lower level of abstraction into $f$ omission failures at the upper level[3]. The basic idea of the resulting tick synchronization algorithm is depicted in Figure 3 and works as follows: The system comprises $2f + 2$ virtual supernodes (representing the upper layer), each of which incorporates $f+1$ subnodes. Each single (sub)node $p.i$, where $p$ denotes the supernode and $i \in \{1, \ldots, f+1\}$ the subnode number, runs the transformation algorithm, i.e., Algorithm 4, together with an instance of Algorithm 3. In Figure 3 virtual supernode $p$ with identifier $2f + 2$, with three of its (sub)nodes $p.1, p.f$ and $p.f+1$, is depicted in large. One can see that $p.1$ comprises of the omission tolerant algorithm together with an instance of the transformation Algorithm 4 depicted as a set of voters vnode $1, \ldots,$ vnode $2f+1$.

Generally, Algorithm 4 located at node $p.i$ waits until tick $k$ has been received from all (sub)nodes $q.1, \ldots, q.f+1$ of a supernode $q$ before it delivers tick $k$ to the omission-tolerant algorithm running at $p.i$.

Each (sub)node $p.i$ constitutes a Byzantine fault containment region of its own and thus may fail arbitrarily. By our failure mode assumption up to $f$ subnodes may suffer from a fault. By waiting for all $f+1$ subnodes of a supernode, Algorithm 4 ensures that tick $k$ is not delivered to the tick synchronization algorithm running at $p.i$ before it was sent by at least one correct subnode, which makes it compliant to Algorithm 3's assumptions.

While Algorithm 3 can be implemented in hardware by analogous means as Algorithm 1, it is not self-evident that a feasible implementation exists for the transformation algorithm. A possible implementation of a single vnode $q$ task is depicted in Figure 4. To overcome the problem of voting over unsynchronized subnodes sending anonymous transitions only, the transformation algorithm needs a possibility to store ticks. Elastic Pipelines can be used for this

[3]To be precise, the Byzantine failures are transformed into late timing failures. However, Algorithm 3 tolerates late timing failures, too.

**Algorithm 4** Transformation algorithm at (sub)node $p.i$

```
for all supernodes $q \neq p$ execute task:
vnode $q$:
1: if received tick(k) from all nodes $q.j$, $1 \leq j \leq f+1$ then
2:     deliver tick(k) from supernode $q$ to node $p.i$;
```

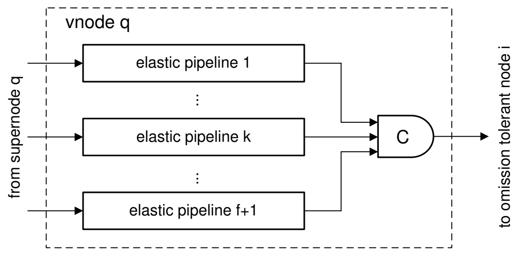

**Figure 4. Implementation of a single vnode of the transformation algorithm.**

**Table 2. Impl. complexities of $\mathcal{S}_{Byz}$ and $\mathcal{S}_{Om}$**

| | nodes $n$ | links $\ell$ |
|---|---|---|
| $\mathcal{S}_{Byz}$ | $3f+2$ | $9f^2+9f+2$ |
| $\mathcal{S}_{Om}$ | $2f^2+4f+2$ | $4f^4+14f^3+18f^2+10f+2$ |

purpose. The $f+1$-input Muller C-Element only generates a tick if it received a tick in each of its preceding pipelines. Note that the pipeline size is bounded by the difference of the number of ticks received from two correct subnodes of the same supernode, i.e., by $\pi$.

In summary, we have bought the advantage of using a simpler algorithm by increasing the number of nodes.

**Comparison of algorithms:** In order to evaluate whether this pays off we compare the two Byzantine tick generation solutions: (a) a distributed system running instances of Algorithm 1, further denoted by $\mathcal{S}_{Byz}$, and (b) the failure transformation based solution with Algorithm 3 and Algorithm 4, abbreviated by $\mathcal{S}_{Om}$, with respect to implementation complexity and performance.

*Implementation Complexity:* To tolerate $f$ Byzantine failures, $\mathcal{S}_{Byz}$ requires $n \geq 3f+2$. The number of links $\ell$ is given by $\ell = n(n-1) \geq 9f^2+9f+2$. In case of $\mathcal{S}_{Om}$, $n' \geq 2f+2$ must hold for the number of supernodes $n'$, i.e., the number of (sub)nodes needs to be $n \geq (2f+2)(f+1) = 2f^2+4f+2$. Since every node receives ticks from the nodes of all supernodes except its own, it requires $(n'-1)(f+1) \geq 2f^2+3f+1$ incoming links, yielding $\ell \geq (2f^2+3f+1)(2f^2+4f+2) = 4f^4+14f^3+18f^2+10f+2$. The results are summarized in Table 2.

At the first sight $\mathcal{S}_{Byz}$ scales far better than $\mathcal{S}_{Om}$ with growing $f$. A closer look, however, reveals that dismissing $\mathcal{S}_{Om}$ is not necessarily the optimal choice since the nodes have different internal complexity. In the following let $A_{Byz}$ resp. $A_{Om}$ be a measure for the implementation complexity of a single node of solution $\mathcal{S}_{Byz}$ resp. $\mathcal{S}_{Om}$. We further denote with $A^{sys}_{Byz}(f)$, resp., $A^{sys}_{Om}(f)$ the complexity of distributed system $\mathcal{S}_{Byz}$ resp. $\mathcal{S}_{Om}$ tolerating $f$ Byzantine failures. Note that different implementation techniques will require different complexity measures. In case of hardware, e.g., the die size is a natural choice for comparison. Obviously this measure is not suited for measuring software complexity and thus comparing complexity can only be done with respect to a given target technology.

Recalling the implementation complexity of Section 5, we obtain $A^{sys}_{Byz}(3) = 11A_{Byz} \approx 5mm^2$ and $A^{sys}_{Om}(3) = 32A_{Om} \approx 1mm^2$ which clearly means that $A^{sys}_{Om}(3) < A^{sys}_{Byz}(3)$. The question arises whether the break-even point expected from the analysis above has not been reached at $f = 3$, such that $A^{sys}_{Byz}(f)$ scales better for larger $f$. It turns out that this is not the case. The reason is that $A_{Byz}$ resp. $A_{Om}$ depend on $f$ and $A_{Om}(f) \ll A_{Byz}(f)$. From Section 4, we know that $A_{Om}(f) \propto \binom{2f+1}{f+1}$ and $A_{Byz}(f) \propto \binom{3f+1}{f+1}$ in case of CMOS technology, because of the THMs used in the algorithms. Figure 5 depicts the trends of $A^{sys}_{Byz}(f)$ and $A^{sys}_{Om}(f)$ for $f \in \{1, \dots, 10\}$. Note that multiplicative factors are set to 1, i.e., $A^{sys}_{Byz} = (3f+2)\binom{3f+1}{f+1}$. Since the implementation complexity scale is logarithmic, a multiplication by a factor would result in a translation of the curve only. Thus, one can clearly see that $A^{sys}_{Om}(f)$ scales far better than $A^{sys}_{Byz}(f)$. In the following we prove that $A_{Byz}(f)$ overwhelms $A_{Om}(f)$ by a factor of $(1+\frac{1}{3})^{f+1}$:

$$\begin{aligned}
\frac{A_{Byz}(f)}{A_O(f)} &= \frac{\binom{3f+1}{f+1}}{\binom{2f+1}{f+1}} \\
&= \frac{(3f+1)\dots(2f+1)}{(2f+1)\dots(f+1)} \\
&= \left(1+\frac{f}{2f+1}\right)\cdots\left(1+\frac{f}{f+1}\right) \\
&\geq \left(1+\frac{f}{2f+1}\right)^{f+1} \\
&\geq \left(1+\frac{1}{3}\right)^{f+1} \qquad \text{if } f \geq 1.
\end{aligned}$$

Thus, if $A^{sys}_{Byz}(f) = (3f+2)A_{Byz}(f)$ and $A^{sys}_{Om}(f) = (2f^2+4f+2)A_{Om}(f)$, the factor of $(1+\frac{1}{3})^{f+1}$ in algorithm complexity per node outweighs the factor of $f$ in the number of nodes by far and renders solution $\mathcal{S}_{Om}$ more scalable than $\mathcal{S}_{Byz}$, i.e., $A^{sys}_{Byz}(f) \geq A^{sys}_{Om}(f)$. In other words, the savings obtained by the simplification of the algorithm outweigh the overheads of the node replication required to justify the simpler algorithm.

Note, however, that although this result is representative for standard CMOS implementations of our algorithms, very different results are obtained if non-standard technologies or software implementations are used. In case of the latter an appropriate complexity measure is code size together with memory usage. Since the code size of the Byzantine tolerant and the omission tolerant algorithm is constant (wrt. $n$ and $f$) and both only have to store the difference of ticks generated locally and remotely, $A_{Byz}(f) = A_{Om}(f) \in \mathcal{O}(n) = \mathcal{O}(f)$. Thus, clearly $A^{sys}_{Byz}(f) \leq A^{sys}_{Om}(f)$.

*Performance:* Clearly solution $\mathcal{S}_{Byz}$ will produce ticks at a higher frequency since the frequency is determined by the minimum interconnection delay $\delta$ between remote nodes. In case of $\mathcal{S}_{Om}$, ticks sent by $p$ and received by $q$ have to pass through an additional pipeline (in front of the voter) of length $\pi$, which increases this delay.

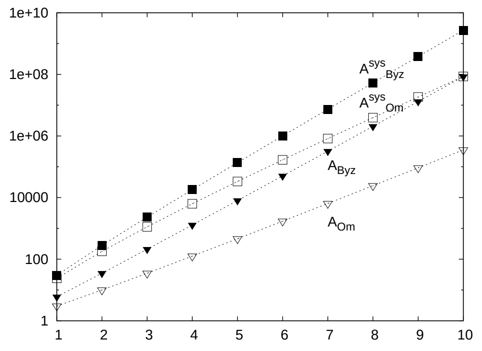

**Figure 5. Implementation complexities** $A_{Byz}(f)$**,** $A_{Om}(f)$**,** $A^{sys}_{Byz}(f)$ **and** $A^{sys}_{Om}(f)$

## 8. Conclusions

The DARTS example demonstrates that SoCs can indeed benefit from algorithms and concepts developed in the distributed systems community. As we have pointed out, however, a one-to-one translation of the existing, software-based implementations to hardware is generally not efficient, since the implementation efforts for certain functions differ significantly. One striking example here is the threshold function, whose mapping to standard digital hardware proved to be extremely costly.

Even more importantly, the traditional failure models used in the distributed community cannot be directly adopted for SoCs: In the light of different hardware faults and different implementation costs and constraints, the tradeoffs from software-implemented distributed systems are no longer valid for SoCs. In particular, we have questioned the appropriateness of using the Byzantine failure model and tried to identify a restricted failure model that better suits the needs of SoCs. It turned out that there is indeed a substantial potential for savings in the implementation, which comes at the price of reduced hardware fault coverage, however.

We have proposed the concept of failure transformation to overcome this dilemma and have demonstrated that its use indeed enables a reduction of the overall system cost. The super-linear dependence of implementation cost on the number of faults to be tolerated can be partly defeated by a careful structuring into virtual super-nodes exhibiting weak failure semantics that are internally composed of sub-nodes with unrestricted failure semantics.

Although the particular tradeoff heavily depends on degree of fault tolerance, target technology and type of algorithm, our case study allows the general conclusion that a careful consideration and particularly a structuring of the failure model can yield substantial savings.

## References

[1] V. Beiu, J. M. Quintana, and M. J. Avedillo. VLSI Implementations of Threshold Logic – A Comprehensive Survey. *IEEE Transactions on Neural Networks*, 14(5):1217–1243, Sept. 2003.

[2] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005.

[3] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, Oct. 1984.

[4] B.-R. Choi, K. Park, and M. Kim. An improved hardware implementation of the fault-tolerant clock synchronization algorithm for large multiprocessor systems. *IEEE Transactions on Computers*, 39(3):404–407, Mar. 1990.

[5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, July 2003.

[6] D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

[7] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[8] S. Dolev and Y. Haviv. Self-stabilizing microprocessors, analyzing and overcoming soft-errors. *IEEE Transactions on Computers*, 55(4):385–399, Apr. 2006.

[9] S. Dolev and Y. Haviv. Stabilization enabling technology. In *Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, LNCS, 2006.

[10] M. Ferringer, G. Fuchs, A. Steininger, and G. Kempf. VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT2006)*, Oct. 2006.

[11] M. Fuegger, U. Schmid, G. Fuchs, and G. Kempf. Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. *Sixth European Dependable Computing Conference (EDCC-6)*, Oct. 2006.

[12] International technology roadmap for semiconductors, 2005.

[13] G. Le Lann and U. Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003.

[14] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[15] A. Maheshwari, I. Koren, and W. Burleson. Accurate estimation of Soft Error Rate (SER) in VLSI circuits. In *Proceedings of the 2004 IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 377–385, Oct. 2004.

[16] M. S. Maza and M. L. Aranda. Analysis of clock distribution networks in the presence of crosstalk and groundbounce. In *Proceedings International IEEE Conference on Electronics, Circuits, and Systems (ICECS)*, pages 773–776, 2001.

[17] A. K. Palit, V. Meyer, W. Anheier, and J. Schloeffel. Modeling and analysis of crosstalk coupling effect on the victim interconnect using the ABCD network model. In *Proceedings of the 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'04)*, pages 174–182, Oct. 2004.

[18] K. Shin and P. Ramanathan. Transmission delays in hardware clock synchronization. *IEEE Transactions on Computers*, 37(11):1465–1467, Nov. 1988.

[19] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[20] I. E. Sutherland. Micropipelines. *Communications of the ACM, Turing Award*, 32(6):720–738, June 1989. ISSN:0001-0782.

[21] D. VanAlen and A. Somani. An all digital phase locked loop fault tolerant clock. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 3170–3173, June 1991.

[22] N. Vasanthavada and P. Marinos. Synchronization of fault-tolerant clocks in the presence of malicious failures. *IEEE Transactions on Computers*, 37(4):440–448, Apr. 1988.

[23] J. Widder. *Distributed Computing in the Presence of Bounded Asynchrony*. PhD thesis, Vienna University of Technology, Fakultät für Informatik, 2004.

[24] J. Widder, G. Le Lann, and U. Schmid. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 20–37, Budapest, Hungary, Apr. 2005. Springer Verlag.