Brief Announcement: How to Speed-up Fault-Tolerant Clock Generation in VLSI Systems-on-Chip via Pipelining

Andreas Dielacher
RUAG Aerospace Austria
andreas.dielacher@ruag.com

Matthias Függer, Ulrich Schmid
Technische Universität Wien
{fuegger,s}@ecs.tuwien.ac.at

Modern *very-large scale integration* (VLSI) circuits, in particular, *systems-on-chip* (SoC), have much in common with the loosely-coupled distributed systems that have been studied by the fault-tolerant distributed algorithms community for decades [1, 4]. Recent work confirms that distributed algorithms results and methods are indeed applicable in VLSI design, and, conversely, that results and methods from VLSI design can also be applied successfully in the distributed algorithms context. Examples of the latter are error-correcting codes and *pipelining*, which is probably the most important paradigm for concurrency in VLSI design.

In general, pipelining is applicable in case of streamed data processing, where a sequence of individual data items is to be processed by a sequence of $x > 1$ actions $a_1, \dots, a_x$, applied to every data item. Consider a chain of processors $p_1, \dots, p_x$ connected via storage elements (buffers), which provide the output of p_{i-1} as an input to p_i. Instead of executing all actions by a single processor sequentially, every action a_i is performed by processor p_i here. Consequently, assuming a stream of data items to be processed, every single data item flows through the chain of processors similar as gas or water flows through a pipeline. Assuming that every action takes α seconds of processing time on its processor, the pipeline can process $1/\alpha$ data items per second. This is a speed-up of a factor x over the at most $1/(x\alpha)$ data items per second that could be digested by a single processor executing all x actions sequentially.

Pipelining is a well-known technique for speeding-up synchronous distributed algorithms. In this paper,[1] we demonstrate that pipelining is also effective for speeding-up asynchronous, fault-tolerant distributed algorithms in systems with large bandwidth$\times$delay products. Basically, the idea is to just exploit the fact that any non-zero delay FIFO end-to-end data transmission/processing path stores the whole information in transit, and hence has an inherently pipelined architecture. A fault-tolerant distributed algorithm may hence immediately start phase k (rather than wait for the acknowledgments of the previous data processing phase $k-1$ in a "stop-and-go fashion"), provided that the acknowledgments for phase $k-x$ (for some integer $x > 1$) have already been received from sufficiently many correct processes. If the system has at least x stages in the inherent pipeline of every end-to-end delay path, with stage delays δ_i and $\delta_{ete} = \sum_{i=1}^{x} \delta_i$, this allows to speed-up the processing throughput from $1/\delta_{ete}$ up to $1/\delta_s$, where s is the slowest stage in the pipeline.

We demonstrate the feasibility of this idea by providing a pipelined version of the DARTS fault-tolerant clock generation approach for SoCs introduced in [3]. Instead of using a quartz oscillator and a clock tree for disseminating the clock signal throughout the chip, DARTS clocks employ a Byzantine fault-tolerant distributed tick generation algorithm (TG-Alg). The algorithm is based on a variant of consistent broadcasting [5], which has been adapted to the particular needs of a VLSI implementation [3]. Unfortunately, since the frequency of an ensemble of DARTS clocks is solely determined by the end-to-end delays along certain paths (which depend on the physical dimensions of the chip and hence cannot be made arbitrarily small), the maximum clock frequency is limited. For example, our first FPGA prototype implementation ran at about 24 MHz; our recent space-hardened 180 nm CMOS DARTS ASIC runs at about 55 MHz. Fortunately, pipelining comes as a rescue for further speeding-up the clock frequency here.

Like DARTS, the pipelined pDARTS derives from a simple synchronizer for the Θ-Model introduced in [5]. Its pseudocode description is given below; X is a system parameter determined by the inherent pipeline depth of the system. The algorithm assumes a message-driven system (where processes, i.e., TG-Alg instances, make atomic receive-compute-send steps whenever they receive a message), where at most f of $n = 3f + 1$ processes may behave Byzantine. Correct processes are connected by a reliable point-to-point message-passing network (= TG-Net), with end-to-end delays within some (unknown) interval $[\tau^-, \tau^+]$. Let $\Theta = \tau^+/\tau^-$ denote the maximum delay ratio.

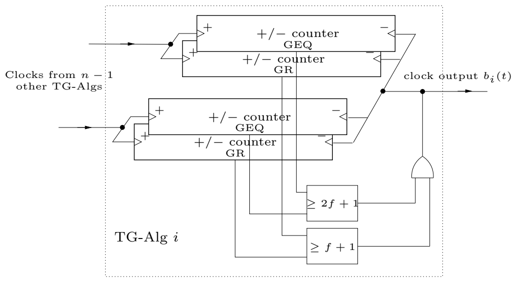
```
1: VAR k: integer := 0      /* Local clock value */
2: send tick($-X$) ... tick(0) to all      /* At booting time */
3: if received tick($\ell$) from $f+1$ processes, with $\ell > k$ then
4:     send tick($k+1$) ... tick($\ell$) to all [once]
5:     $k := \ell$
6: if received tick($\ell$) from $2f+1$ processes, with $\ell \geq k - X$ then
7:     send tick($k+1$) to all [once]
8:     $k := k+1$
```

Our detailed analysis revealed that correct processes generate a sequence of consecutive messages tick(k), $k \geq 1$, in a synchronized way: If $b_p(t)$ denotes the value of the variable k of the TG-Alg at process p at real-time t, which gives the number of tick(ℓ) messages broadcast so far, it turns out that $(t_2 - t_1)\alpha_{\min} \leq b_p(t_2) - b_p(t_1) \leq (t_2 - t_1)\alpha_{\max}$ for any correct process p and $t_2 - t_1$ sufficiently large ("accuracy"); the constants $\alpha_{\min}$ and $\alpha_{\max}$ depend on τ^-, τ^+ and X. More-

[1]This work has been supported by the FIT-IT project DARTS (809456) and the FWF projects P17757 and P20529.

over, every two correct processes maintain $|b_p(t) - b_q(t)| \leq \pi$ ("precision"), for a small constant π that depends on Θ and X only.

Since the pDARTS algorithm looks very simple, it is tempting to conclude that it is easily implemented in hardware. It turned out, however, that several challenging issues must be solved to actually accomplish this:
How to circumvent unbounded tick(k), $k \in \mathbb{N}$? Clearly, a VLSI distributed algorithm that broadcasts unbounded integers is infeasible. We hence modified the algorithm to only broadcast event-messages (up/down signal transitions), and maintain the number ℓ_q of event-messages seen from TG-Alg q so far at the receiver. Note that this results in a TG-Net formed by feeding all clock signals to all TG-Algs (e.g., by a bus). This evidently avoids unbounded message size, albeit not unbounded size of the variable ℓ_q. Fortunately, the inspection of the catchup rule (line 3) and the progress rule (line 6) reveals that they only check, for sufficiently many ($f+1$ and $2f+1$) q, whether $\ell_q - k > 0$ and $\ell_q - k \geq -X$, respectively. These differences, however, always remain bounded due to bounded precision π.
How to ensure atomicity of actions in an asynchronous VLSI implementation? Distributed computing models assume atomic computing steps at the level of a single process. This abstraction does not apply when the algorithm is implemented via asynchronous digital logic gates, which concurrently and continuously compute their outputs based on their inputs. Thus, explicit synchronization, i.e., serialization of actions via handshaking, or implicit synchronization via timing constraints must be employed if two local computations must not interfere with each other (namely, line 3 and line 6 for the same tick(k), cf. "[once]" in the TG-Alg).



The major building blocks of a single pDARTS TG-Alg are $2(n-1)$ +/− *counters*, two for each of the $n-1$ other TG-Algs q in the system. The two +/− counters for q are responsible for maintaining $\ell_q - k > 0$ (resp. $\ell_q - k \geq -X$), and provide the binary status signals GR (resp. GEQ) that signals when the inequality holds. In addition, a "$\geq f+1$" (resp. "$\geq 2f+1$") *threshold circuit* implements the rules in line 3 (resp. line 6). Finally, there is a device (shown as an OR-gate), which is responsible for generating every local clock tick exactly once from the outputs of the threshold gates.
How to disambiguate GR, GEQ for different ticks? The major problem encountered when implementing the pDARTS algorithm is the lack of atomicity in any asynchronous digital circuit. First, we had to split e.g. the signal GR into two signals GR^e and GR^o, tied to even and odd ticks, respectively: GR^e is true iff the inequality $\ell_q - k > 0$ holds and $k \in \mathbb{N}_{even} := 2\mathbb{N}$. The associated threshold circuit is also duplicated, and all their outputs are combined in some straightforward way to finally generate p's odd and even ticks. The problem of avoiding the mixing-up of GR^e signals based on, say, $k=2$ and $k=4$, is solved implicitly by means of a timing constraint, which allowed us to prove a pivotal "interlocking property".

The formal analysis of pDARTS is based on a novel modeling and analysis framework, which, unlike classic trace theory, applies to hardware-implemented asynchronous *fault-tolerant* distributed algorithms. It deals with a hierarchy of *modules*, which possess input and output *ports* (binary signals). A module's behavior specifies how the signals on the input and output ports are related. Note that modules differ from timed automatons primarily in that they *continuously* compute their outputs, based on the history of their inputs. *Composite modules* consist of multiple sub-modules and their interconnect, which specifies how sub-module ports are connected to each other and to the module's input/output ports. Note that the interconnect specification itself assumes zero delays; modeling non-zero interconnect delays, e.g., for real wires, requires intermediate channels: A channel possesses a single input port and a single output port, and its behavior specifies bounded delay FIFO delivery of input port signal transitions at the output port. Modules that are not further refined are called *basic modules*. Elementary basic modules are zero-delay boolean functions (AND, OR, ...) and channels.

The behavior of a (non-faulty) composite module is determined by the behavior of its constituent sub-modules; the behavior of a basic module must be given a priori. An execution of a system is specified by the behaviors of each of its signals, and is typically modeled as a set of event traces. Correctness proofs establish properties of the behaviors of the algorithm's physical implementation, based on the assumption that (1) the system and failure model holds, and (2) that the implementations of the basic modules indeed satisfy their behavioral specifications. As this requires verification/falsification of (1) and (2) only, we avoid the substantial "proof gap" of standard (high-level) distributed computing models.

Using this framework, we provided a specification of the [fairly simple] pDARTS basic modules that make up e.g. the +/− counters, and derived a comprehensive correctness proof and worst case performance analysis of the pDARTS algorithm. The validity of our results was confirmed by measurement results obtained in an FPGA prototype system, which revealed that pipelining indeed allows to entirely remove the dependency of the local clock frequency on large delays in the TG-Net.

All details can be found in the full version [2] of our paper.

1. REFERENCES

[1] B. Charron-Bost, S. Dolev, J. Ebergen, and U. Schmid, editors. *Fault-Tolerant Distributed Algorithms on VLSI Chips*, Schloss Dagstuhl, Germany, Sept. 2008.

[2] A. Dielacher, M. Fuegger, and U. Schmid. How to Speed-up Fault-Tolerant Clock Generation in VLSI Systems-on-Chip via Pipelining, RR 15/2009, TU Wien, Inst. f. Technische Informatik, 2009.

[3] M. Fuegger, U. Schmid, G. Fuchs, and G. Kempf. Fault-tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In *Proc. EDCC-6*, Coimbra, Portugal, p. 87–96, Oct. 2006.

[4] U. Schmid. Keynote: Distributed algorithms and VLSI. *SSS'08*, Detroit, USA, Nov. 2008.

[5] J. Widder and U. Schmid. The Theta-Model: Achieving synchrony without clocks. RR 49/2005, TU Wien, Inst. f. Technische Informatik, 2005. (To appear in Distributed Computing, 2009).