Journal of Computer and System Sciences 80 (2014) 860–900

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Rigorously modeling self-stabilizing fault-tolerant circuits: An ultra-robust clocking scheme for systems-on-chip ☆

Danny Dolev [a,1], Matthias Függer [b], Markus Posch [b], Ulrich Schmid [b], Andreas Steininger [b], Christoph Lenzen [c,*,2]

[a] School of Engineering and Computer Science, The Hebrew University of Jerusalem, Edmond Safra Campus, 91904 Jerusalem, Israel
[b] Department of Computer Engineering, Vienna University of Technology, Treitlstrasse 3, 1040 Vienna, Austria
[c] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 32 Vassar Street, 02139 Cambridge, MA, USA

ARTICLE INFO

Article history:
Received 23 January 2013
Received in revised form 18 October 2013
Accepted 6 January 2014
Available online 15 January 2014

Keywords:
Modeling framework
Clock synchronization
Hardware implementation
Experiments
Metastability
Dependability
Theoretical analysis
Hybrid state machines
Byzantine fault-tolerance
Self-stabilization

ABSTRACT

We present the first implementation of a distributed clock generation scheme for Systems-on-Chip that recovers from an unbounded number of arbitrary transient faults despite a large number of arbitrary permanent faults. We devise self-stabilizing hardware building blocks and a hybrid synchronous/asynchronous state machine enabling metastability-free transitions of the algorithm's states. We provide a comprehensive modeling approach that permits to prove, given correctness of the constructed low-level building blocks, the high-level properties of the synchronization algorithm (which have been established in a more abstract model). We believe this approach to be of interest in its own right, since this is the first technique permitting to mathematically verify, at manageable complexity, high-level properties of a fault-prone system in terms of its very basic components. We evaluate a prototype implementation, which has been designed in VHDL, using the Petrify tool in conjunction with some extensions, and synthesized for an Altera Cyclone FPGA.

1. Introduction & related work

In the past, computers have essentially been viewed as monolithic, synchronous, fault-free systems. If at all, fault-tolerance has been introduced (i) to deal with limited, specific failures (e.g. errors in communication or data read from storage, which are usually handled via error-correcting codes), and (ii) at the level of distributed systems comprised of multiple machines that are fault-prone or subject to attacks (e.g. data centers or peer-to-peer applications, which use some form of replication). Except for critical systems and extreme operational conditions (e.g. medical or aerospace applications [1]),

☆ This work has been supported in part by the Austrian Science Foundation (FWF) project FATAL (P21694) and SIC (P26436), by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), by the ISG (Israeli Smart Grid) Consortium, administered by the office of the Chief Scientist of the Israeli Ministry of Industry and Trade and Labor, and by grant 3/9778 of the Israeli Ministry of Science and Technology.
* Corresponding author.
E-mail addresses: dolev@cs.huji.ac.il (D. Dolev), fuegger@ecs.tuwien.ac.at (M. Függer), markus.posch@gmx.at (M. Posch), s@ecs.tuwien.ac.at (U. Schmid), steininger@ecs.tuwien.ac.at (A. Steininger), clenzen@csail.mit.edu (C. Lenzen).
[1] Danny Dolev is incumbent of the Berthold Badler Chair.
[2] Christoph Lenzen has been supported by the Swiss National Science Foundation (SNF), the Swiss Society of Friends of the Weizmann Institute of Science, and the German Research Association (DFG, reference number Le 3107/1-1) for this work.

http://dx.doi.org/10.1016/j.jcss.2014.01.001
0022-0000/

there has been little motivation to build systems that are robust on all levels from scratch, a process that involves redesigning—or even reinventing—the very basics of how computations are organized and performed.

Due to the tremendous advances of *Very Large Scale Integration* (VLSI) technology, this situation has changed. Enabled by ever decreasing feature sizes and supply voltages, modern circuits nowadays accommodate billions of transistors running at GHz speeds [2]. As a consequence, the assumption of chip-global (not to speak of system-global) synchrony [3] and no (or restricted) faults gradually became outdated [4]. Improved process technology and architectural-level fault-tolerance measures are common nowadays, and the lack of global synchrony has been tackled by accepting a certain level of asynchrony between different parts of the system.

In the most extreme form of this approach, computations are completely unsynchronized at all levels [5], which requires to synchronize all dependent activities (like sending and receiving of data) explicitly via handshaking. In contrast, *Globally Asynchronous Locally Synchronous (GALS)* systems [6] make use of local clock sources to drive synchronous computations within each *clock domain*. Note that, in the wider sense, most multiprocessors fall into this category, as there is usually no single common clock that drives all processors. GALS systems again can be divided into two general classes: One that operates asynchronously at the inter-domain level, and the other consisting of *multi-synchronous* systems [7,8] that provide some, albeit reduced, degree of synchronization among clock domains. The former class suffers from the drawback that, for inter-domain communication, either strong synchronizers or stoppable clocks must be foreseen [9]. After all, every bit of the sender's data must have stabilized at the receiver before the clock edge used for reading the data occurs. This is avoided in multi-synchronous systems, where high-speed inter-domain communication via FIFO buffers can be implemented due to the available global synchronization [10]. Since the latter abstraction is also very useful for other purposes, multi-synchronous GALS is preferable from the viewpoint of a system-level designer.

Naturally, establishing inter-domain synchronization comes at additional costs. While it is not too difficult to achieve and maintain in the absence of faults [11,12], the issue becomes highly challenging once faults of clocking system components enter the picture.

1.1. Contribution

We present an FPGA prototype implementation of a distributed clock generation scheme for SoC that self-stabilizes in the presence of up to $f < n/3$ faulty nodes. It incorporates the pulse algorithm from [13] that tolerates arbitrary clock drifts and allows for deterministic recovery and (re)joining in constant time if $n - f$ nodes are synchronized; it stabilizes within time $\mathcal{O}(n)$ with probability $1 - 2^{-n}$ from any arbitrary state. An additional algorithmic layer that interacts weakly with the former provides bounded high-frequency clocks atop of it. Nodes executing the compound algorithm broadcast a mere constant number of bits in constant time. The formal proofs of the properties of the pulse synchronization algorithm and the derived high-frequency clocks are given in [13].

Deriving an implementation from the specification of the algorithm in [13] proved to be challenging, as the high-level theoretical model and formulation of the algorithm in [13] abstracts away many details. Firstly, it assumes a number of basic self-stabilizing modules above the level of gates and wires to be given. We devise and discuss self-stabilizing implementations of these building blocks meeting the specifications required by the high-level algorithm. Secondly, the algorithm's description is in terms of state machines performing transitions that are non-trivial in the sense that they do not consist of switching a single binary signal or memory bit. This requires careful consideration of metastability issues, since these state transitions are triggered by information from different clock domains. In order to resolve this issue, we introduce a generic *Hybrid State Transition Machine (HSTM)* that asynchronously starts a local synchronous execution of a state transition satisfying the model specification from [13]. Related to this matter, we thirdly discuss in detail how the algorithm and its implementation make a best effort to guard against metastable upsets. Here, we try to get the best out of the design decisions and rely on synchronizers only where absolutely necessary.

These non-trivial implementation issues and the complex interactions between the basic building blocks raise the question under which circumstances the high-level properties of the algorithm shown in [13] indeed hold for the presented implementation. To answer this question, we devised a model that is able to capture the behavior of the constructed modules, including faults, resilience to faults, and self-stabilization, in a hierarchical fashion. By specifying the desired behavior of modules in terms of the feasible output generated in response to their inputs, we can also *reason* about the behavior of (implementations of) modules in a hierarchical manner. This property is crucial, as it permits to determine conditions under which our implementation indeed satisfies the requirements by the abstract model used in [13], and then soundly conclude that if these conditions are met, all statements made in [13] apply to our implementation. Since our approach is highly generic and permits to adjust the granularity of the description in order to focus on specific aspects of the system, we believe it to be of general and independent interest in the context of devising fault-tolerant systems.

In order to verify the predictions from theory,[3] we carried out several experiments incorporating drifting clocks, varying delays, and both transient and permanent faults. This necessitated the development of a testbed that can be efficiently controlled and set up for executing a large number of test runs quickly. In our 8-node prototype implementation, the

[3] Or, to be scientifically accurate, we rather successfully failed at falsifying them. Our implementation primarily serves as a proof of concept, as clearly an FPGA implementation can merely hint at the properties of an ASIC.