Efficient Construction of Global Time in SoCs despite Arbitrary Faults

Christoph Lenzen
Massachusetts Institute of Technology
Cambridge, MA, USA
clenzen@csail.mit.edu

Matthias Függer, Markus Hofstätter, Ulrich Schmid
Vienna University of Technology
Vienna, Austria
{fuegger,mhofstaetter,s}@ecs.tuwien.ac.at

***Abstract*—In this paper, we show how to build synchronized clocks of arbitrary size atop of existing small-sized clocks, despite arbitrary faults. Our solution is both self-stabilizing and Byzantine fault-tolerant, and needs merely single-bit channels. It involves a reduction to Byzantine fault-tolerant consensus, which allows different consensus algorithms to be plugged in for matching the actual clock sizes and resilience requirements best. We demonstrate the practicability of our approach by means of an FPGA implementation and its experimental evaluation. To also address the cases where deterministic algorithms hit fundamental limits, we provide a novel randomized self-stabilizing Byzantine consensus algorithm that works very well also in these settings, along with its correctness proof and stabilization time analysis.**

I. Introduction

We address the problem of how to construct a common notion of time among the subsystems of a System-on-Chip (SoC), the processors of a multicore, or the nodes of any other "hardware-level" distributed system, in the presence of *arbitrary* failures. The ability to trigger operations at well-defined times at different nodes, i.e., to globally coordinate activities and timestamp data, greatly simplifies the design of distributed applications. Indeed, using *time instead of timeouts* [1] has not only been the dominant paradigm for implementing replicated state-machines in fault-tolerant distributed computing since decades, but has also made its way into *Network-on-Chip* (NoC) [2] and SoC [3] technology.

Establishing a common notion of time is simple in the absence of failures: A sufficiently wide (say, 32-bit) data bus that continuosly feeds the current time, maintained by a single counter driven by some oscillator, to all nodes that need access to global time does the job. If a common clock signal is available at every node, this global data bus can also be replaced by *local* counters that are incremented synchronously at all nodes. Unfortunately, however, these approaches are not fault-tolerant: A global data bus is a single point of failure, as is a common clock signal. In addition, the use of local counters may turn transient failures (e.g. caused by single-event upsets of the counters due to ionizing particles [4]) into permanent ones, unless a recovery mechanism is in place.

To alleviate this problem, all local counters must be regularly checked for time offsets and, if needed, adjusted to match the correct time. As surveyed below, distributed computing research has provided appropriate solutions both for classic Byzantine fault-tolerant distributed systems and for self-stabilizing systems. Byzantine fault-tolerant solutions assume that some fraction of the nodes (usually at most $1/3$) may behave arbitrarily, in the sense that they can send anything to the other nodes in the system. By contrast, classic self-stabilizing solutions can recover from an arbitrary corruption of the state of *every* node, provided that no further failure occurs during the stabilization period. Byzantine fault-tolerant self-stabilizing solutions combine the best of both worlds. They completely mask transient faults up to the fault-tolerance threshold and recover automatically after massive transient failures, e.g., in spacecraft electronics after a solar flare. Unfortunately, however, existing algorithms either incur features that make them unsuitable in our context or, at best, allow the implementation of small-sized synchronized clocks only.

Contribution

In this paper, we show how to maintain large (say, 32 or 64 bit) local counters at every node that extend an already available synchronized short clock (say, 4-8 bits wide), such that their catenation can be used as a *local clock* that provides access to a wide-range global time. More specifically, for a fully-connected distributed system of n nodes (we expect n to be typically in the range of $4 \dots 16$), among at most $f = \lfloor (n-1)/3 \rfloor$ can be Byzantine faulty, we will present fault-tolerant self-stabilizing algorithms solving the following problem, which we call (λ, l)-*round-labeling*: Assuming that a (self-stabilizing) synchronized λ-bit clock is already available at every node, extend it to a synchronized $(\lambda + l)$-bit local clock. In order to reduce the wiring complexity, our solutions assume only B-bit channels (typically with $B \ll l$, often just $B = 1$) between every pair of nodes.

Our solution is based on a "generic" reduction of (λ, l)-round-labeling to synchronous binary Byzantine agreement (consensus) [5]. Consensus algorithms with different round complexity resp. resilience can thus be plugged into our solution to match the parameters λ resp. f best. We exemplify this

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-AF-0937274, CNS-1035199, 0939370-CCF and CCF-1217506, the AFOSR under Contract No. AFOSR Award number FA9550-13-1-0042, the Swiss Society of Friends of the Weizmann Institute of Science, the German Research Foundation (DFG, reference number Le 3107/1-1), and the Austrian Science Foundation (FWF) project FATAL (P21694).

by means of the Phase King and Phase Queen algorithms from [6], which can be used as long as $\lambda \geq \log(\lceil 2/B \rceil + 2f + 3)$. A prototype implementation, synthetisized for an FPGA, demonstrates that the resulting algorithm can indeed be implemented easily and works very well in practice. Moreover, we also provide a novel randomized self-stabilizing Byzantine fault-tolerant consensus algorithm, along with its correctness proof and stabilization analysis. It can operate with $B = 1$ and constant λ, i.e., independently of f. It provides a stabilization time (almost) linear in $n + l$, even in the presence of a strong adversary that monitors all communication and states.

Paper Organization

After an overview of related work in Section II, Section III introduces our system model and a generic solution for the (λ, l)-round-labeling problem. Essentially, it reduces the problem to binary consensus on the l bits of the round labels. Section IV resp. Section V provides the analytic resp. experimental results obtained by plugging in the well-known deterministic phase king consensus algorithm into our generic solution. Section VI utilizes a novel randomized consensus algorithm for this purpose, which allows to overcome the inherent limitations of deterministic algorithms w.r.t. feasible values of λ. We conclude in Section VII.

II. Related work

We are not aware of any published solution for the (λ, l)-round labeling problem. It is apparent, however, that the classic clock synchronization problem can be recast as $(0, l)$-round labeling: In case of Byzantine fault-tolerant clock synchronization protocols [7]–[11], nodes are equipped with a priori unsynchronized (l-bit) local clocks that are periodically synchronized with each other. Still, a sufficiently large "core" of correct nodes must *always* be up and running here to maintain synchrony and to correctly integrate late joiners. [12], [13] do not rely on local clocks but rather synchronously increment local counters, triggered by the continuous interaction of the core nodes. [9] and [11] consider the problem for highly specialized system architectures.

Self-stabilizing "digital clock synchronization" algorithms can recover from arbitrarily corrupteded states. However, existing algorithms incur features that make them unsuitable in our (resource-)constrained context. Translating existing solutions [13]–[16] to our setting would (i) also require the availability of synchronized (few-bit) clocks in order to transmit larger messages over B-bit channels, and (ii) result in (expected) stabilization times that grow fast in terms of the system size, i.e., at least quadratic in n. Furthermore, except for the randomized algorithm from [14] (with stabilization time exponential in n), the complexity of local computations would entail a large gate and area consumption. The approach described in [17] does not suffer from a large stabilization time, but relies on very specific assumptions on the system's behavior and is not applicable for $f > 1$ [18]. Moreover, to establish clocks (instead of anonymous pulses), it relies on consensus, implying that again (i) applies. These issues are avoided by the algorithms described in [19], [20], which, however, exhibit a stabilization time that is exponential in the number of clock bits; they are thus suitable for providing small-sized clocks only.

The (λ, l)-round-labeling problem is also related to *recovery* of failed nodes in distributed systems at run-time. In order to do so, enough state information must be transferred to a joining node to build an internal state that is consistent with the current system state. Whereas this is easy when the system is allowed to cease normal operation during such a state transfer (as in rollback-recovery [21]), *transparent* recovery, i.e., synchronizing a joining node with a continuously evolving state on-line is more difficult [22]. We are not aware of solutions that guarantee transparent recovery despite a fraction of the nodes being Byzantine faulty, however, not to speak of approaches that also work with 1-bit channels.

III. System Model and Generic Solution

We consider a fully-connected distributed system of n nodes, among $f < n/3$ are Byzantine faulty. Every node $i \in V = \{1, \ldots, n\}$ is connected to every other node by means of a dedicated B-bit channel, typically with $B = 1$, which allows nodes to exchange B-bit *messages*.[1] Nodes do not need unique identifiers, but they must be able to distinguish senders, e.g., by fixed local communication ports. We do *not* assume that (correct) nodes start executing simultaneously, from some well-defined initial state, at some time origin $t = 0$. Rather, we assume that there is some unknown time t_0, after which all correct nodes faithfully execute the code of their algorithm. The state (local variables etc.) from which every node starts is completely arbitrary, though. Note carefully that this assumption allows massive transient failures to (repeatedly) wipe out the state of the entire system; our algorithms will nevertheless be able to recover from such an event after some short stabilization time.

Every node is also equipped with a discrete-valued *short clock* $C_i(t)$ of length λ bits, which is incremented $(\mathrm{mod}\ 2^\lambda)$ in perfect synchrony[2] at all correct nodes after t_0, but can behave arbitrarily before that time. In the extreme case $\lambda = 0$, this assumption merely implies the existence of a common clock signal. Note carefully, though, that it is outside the scope of this work (in fact, irrelevant) how the short clocks are implemented. Ideally, one would use a self-stabilizing pulse generation algorithm (like, e.g., in [19]) for this purpose.

To retain compatibility with the distributed computing research results we will be using, the time interval between two consecutive clock ticks is called a (lock-step) *round* in the sequel: We assume that, after time t_0, every B-bit message sent by a correct node at the beginning of a round is received by every correct receiver by the end of the same round.

We can now specify the problem we want to solve:

[1] Recast in NoC terminology, we assume a flit size equal to the phit size = B, i.e., we send only B bits per cycle (= round in our terminology).

[2] This assumption is only made for the ease of presentation. It is straightforward to relax this assumption to clocks with bounded imprecision $\pi \geq 0$, as outlined in Section V.

Algorithm 1: (λ, l)-round labeling algorithm at node $i \in V$. $\mathcal{P}$ is a consensus protocol.

```
1 while true do
2     (c_i, b_i) := red(L(i)) // reduction
3     o_i := P(b_i) // run consensus
4     if o_i = true then L(i) := c_i // agreed on c_i
5     else L(i) := 0 // agreed on default
6     wait until round number modulo 2^λ is 0
7     L(i) := L(i) + 1 mod 2^l // increase clock
```

Problem III.1 ((λ, l)*-round labeling*)**.** *Every correct node $i \in V$ maintains an l-bit variable called* round label $L(i)$*, the concatenation of which with the short clock $C_i(t)$ provides a* logical clock $L_i(t)$*. For every pair of correct nodes $i, j \in V$, after some bounded stabilization time after t_0, it holds that $L_i(t) = L_j(t)$ and that both logical clocks are incremented $(\mathrm{mod}\ 2^{\lambda+l})$ in perfect synchrony at the end of every round.*

Our generic solution, shown in Algorithm 1, reduces (λ, l)-round labeling to binary consensus. More specifically, it breaks down the problem to (i) the determination of a candidate label (done via Algorithm 2) followed by (ii) reaching agreement (via some suitable binary consensus algorithm). Recall that consensus is characterized by the following properties [5]:

Agreement: Non-faulty nodes output the same value.
Validity: The output is input of a non-faulty node.
Termination: Non-faulty nodes eventually terminate.

In some more detail, Algorithm 1 works as follows: Whenever the short clock of node i reads $C_i(t) = 0$, an iteration of the while loop starts. It consists of (i) determining a candidate label L and (ii) reaching consensus on whether to set $L(i) := L$ or to reset $L(i) := 0$. Note that Algorithm 1 is indeed generic, in the sense that we can plug in any consensus protocol that works with 1-bit channels, provided it terminates sufficiently fast for both Algorithm 2 and the protocol to be run between two consecutive wrap-arounds of the clocks C_i.

Algorithm 2 shows the other major building block red$(L(i))$ of Algorithm 1: At node i, it takes the current round label $L(i)$ as input and outputs a candidate label c_i and a flag b_i that signals agreement on the candidate label. More specifically, for each bit of the round labels $L(i)$, it determines whether there is an overwhelming majority ($\geq n - f$) for a (unique) value of this bit in the current labels of all nodes in the system. If this is the case for all l bits, a suitable candidate label has been determined; otherwise, the candidate label is set to 0 (= reset). In a second phase, an attempt is made to convince nodes that did a reset of a non-zero candidate. This works for sure if at least $n - f$ nodes have selected the candidate in the first phase, in which case $b_i =$ ***true*** will be returned. Otherwise, $b_i =$ ***false*** can be output, with or without c_i being equal to the candidate label.

The following Theorem III.2 provides the major properties guaranteed by Algorithm 2. Most importantly, it guarantees that there is at most one candidate label L (or 0) system-wide, and that if just one correct node returns $b_i =$ ***true***, then all correct processes agree on the same candidate label.

Algorithm 2: Algorithm red$(L(i))$: reduces (λ, l)-round labeling to binary consensus.

```
   input : leading l bits L(i) of i's logical clock
   output: candidate clock value c_i
           Boolean trust value b_i
1  c_i := L(i) // holds candidate clock value
2  for j ∈ {1, ..., l} do
3      broadcast c_i(j) // j^th bit of candidate value
4      if received ≥ n − f times value c then
5          c_i(j) := c // can happen for only one c
6      else
7          c_i := 0 // input values differ, default to 0
8          break
9  sleep until end of round l // maintain signal 0
10 broadcast(c_i ≠ 0)
11 store set S_i of nodes that sent true
12 b_i := true
13 for j ∈ {1, ..., l} do
14     if c_i ≠ 0 then broadcast c_i(j)
15     if received ≥ n − f times c from S_i then
16         c_i(j) := c // all others see ≥ f + 1 times c
17     else if received ≥ f + 1 times c from S_i then
18         b_i := false // input values differ
19         c_i(j) := c // candidate bit still known
20     else b_i := false // inputs differ, known to all
21 return (c_i, b_i)
```

Theorem III.2. *Using 1-bit channels, Algorithm 1 can be executed in $2l + 1$ rounds. For non-faulty nodes i:*

(1) $\exists L\, \forall i : c_i \in \{0, L\}$ *after the first for-loop.*
(2) $(\forall i : b_i =$ ***false***$)$ *or* $(\forall i : c_i = L)$.
(3) *If* $\exists L \neq 0\, \forall i : L(i) = L$*, then* $\forall i : c_i = L$ *and* $b_i =$ ***true***.
(4) *If* $\forall i : L(i) = 0$*, then* $\forall i : c_i = 0$ *and* $b_i =$ ***false***.

Proof: The time complexity follows by adding the $2l$ rounds required by the two for-loops and the single broadcast round in between.

To show property (1), assume that some non-faulty node i leaves the first for-loop with $c_i = L \neq 0$, as otherwise the statement is trivially satisfied. Thus, it must hold that in each round $j \in \{1, \ldots, l\}$, i received at least $n - f$ times the value $c_i(j)$ (which need not coincede with its own input). Hence, at least $n - 2f \geq f + 1$ non-faulty nodes sent $c_i(j)$ in this round. Consequently, if any non-faulty node i' sets $c_{i'}(j) \neq c_i(j)$ in this round, it received at most $n - f - 1$ times $c_{i'}(j)$ and therefore executes the else-statement in the loop, setting $c_{i'} := 0$ and leaving the loop.

Regarding property (2), suppose that some non-faulty node i leaves the second for-loop with $b_i =$ ***true***. Thus, in each round of the loop, it executed the if-statement (Line 16). It follows that there are at least $n - 2f \geq f + 1$ non-faulty nodes $i' \in S_i$, each of which satisfies $c_{i'} \neq 0$ after the first for-loop. By property (1), we conclude that there is some $L \neq 0$ with $c_{i'} = L$ for all nodes in S_i. In the j^{th} round of the second for-loop, where $j \in \{1, \ldots, l\}$, the at least $f + 1$ non-faulty nodes that broadcasted 1 in the intermediate round broadcast the j^{th} bit of L. Since there are at most f faulty nodes, no node i will

receive more than f times a value different from the j^{th} bit of L from a node in S_i in the j^{th} round of the second loop. Hence, each non-faulty node will execute Line 16 or Line 19 with respect to the j^{th} bit of L in round j of the loop.

The remaining two properties can be verified easily by observing that if all inputs are identical, all at least $n-f$ non-faulty nodes i leave the first for-loop with $c_i = L$, then either all broadcast 1 (if $L \neq 0$) or all broadcast 0 (if $L = 0$); thus, either always all execute Line 16 or Line 20 in the second for-loop, resulting in output $(L, \textbf{\textit{true}})$ or $(0, \textbf{\textit{false}})$, respectively. ■

Note that there are two (orthogonal) modifications of Algorithm 2 that could be used for decreasing the running time. First, if B-bit channels with $B > 1$ are available, the nodes can transmit blocks of B bits in each of the for-loops—instead of just a single bit—and apply the threshold conditions blockwise. This will reduce the running time to $2\lceil l/B \rceil + 1$ rounds. Second, local computations may be fast in comparison to communication delays. In this case, we can speed up the algorithm by means of pipelining: If it is possible to send a bit over a channel every τ time, the for-loops each can be executed in $\ell\tau$ time. Denoting by d the (maximal) communication delay and assuming that the clock resolution is appropriate, the algorithm then can be executed in $3d + 2l\tau$ time. In both cases, Theorem III.2 continues to hold.

Equipped with Theorem III.2, it is not too difficult to prove the correctness and determine the stabilization time of the generic Algorithm 1, as given in the following Theorem III.3:

Theorem III.3. *Suppose that $\mathcal{P}$ is a deterministic r-round consensus protocol and that $\lambda \geq \log(2\lceil l/B \rceil + r + 1)$. Then Algorithm 1 solves (λ, l)-round labeling and stabilizes within $2 \cdot 2^\lambda$ rounds.*

Proof: By the prerequisites and Theorem III.2, the code in the while-loop will execute in $2\lceil l/B \rceil + r + 1$ rounds and therefore can be controlled by a clock with wrap-around every 2^λ rounds. Less than 2^λ rounds after round t_0, it will thus be executed again. Hence the theorem follows if we can show that (i) after a complete execution of the while-loop all values $L(i)$ at non-faulty nodes i are identical, and (ii) if they are identical at the beginning of a complete execution of the loop, they do not change except in its last line.

For both statements, we distinguish two cases. The first case for (i) is that the call to Algorithm 2 returns $b_i = \textbf{\textit{false}}$ at all non-faulty nodes i. Hence, by validity, the call to $\mathcal{P}$ returns ***false*** at all non-faulty nodes which therefore set $L(i) := 0$. On the other hand, if $b_i = \textbf{\textit{true}}$ for some non-faulty node i, Theorem III.2 states that there is some value L satisfying that $c_i = L$ for all non-faulty nodes i. By the agreement property, it follows that either all non-faulty nodes set $L(i) := L$ after executing $\mathcal{P}$ or they all set $L(i) := 0$, establishing (i).

With respect to (ii), suppose that $L(i) = L$ for all non-faulty nodes at the beginning of an iteration of the while-loop. In case $L = 0$, Theorem III.2 states that the call to Algorithm 2 results in $b_i = \textbf{\textit{false}}$ for all non-faulty nodes i; by validity, $\mathcal{P}$ returns ***false*** at all non-faulty nodes which "set" $L(i) := 0$. On the other hand, if $L \neq 0$, the last statement of Theorem III.2 yields that Algorithm 2 returns $(L, \textbf{\textit{true}})$ at all non-faulty nodes. By validity, $\mathcal{P}$ thus must output ***true*** at all non-faulty nodes. Non-faulty nodes will hence "set" $L(i) := L$, concluding the proof. ■

Algorithm 3: Phase King protocol at node $i \in V$.

```
   input: binary value b_i
1  for j ∈ {1,...,f+1} do
2      broadcast b_i // we want to identify a candidate value
3      if received at least n − f times b then
4          broadcast 1b // announce candidate value
5      else broadcast 00 // no unique candidate value identified
6      if received at least n − f times 1b then
7          b_i := b
8          if i = j then broadcast b_i // king's broadcast
9      else
10         if i = j then
11             // king's broadcast
12             if received at least f + 1 times 01 then
13                 broadcast 1 // nodes with value 0 will follow
14             else broadcast 0 // nodes with value 1 will follow
15         if received b from j then b_i := b
16 return b_i
```

IV. Appointing Phase Kings and Queens

In [6], two deterministic consensus algorithms that work with 1-bit channels are introduced, which are perfectly suited for our purposes: The *Phase King* protocol [23] shown in Algorithm 3 achieves optimal resilience ($n \geq 3f + 1$) and requires $4(f + 1)$ rounds with 1-bit channels (and $3(f + 1)$ rounds if 2-bit channels were available). The *Phase Queen* protocol requires $n \geq 4f + 1$ but needs only $2(f + 1)$ rounds.

Plugging in the phase-king algorithm into our generic (λ, l)-round labeling solution Algorithm 1 while recalling Theorem III.3 yields the following result:

Corollary IV.1. *With 1-bit resp. B-bit channels. $B > 1$ and sufficiently large λ, (λ, l)-round labeling can be solved for $\lambda \geq \log(2(2l + 4f + 5))$ resp. $\lambda \geq 2(2\lceil \log(l/B) \rceil + 3f + 4))$ if $n \geq 3f + 1$. For $n \geq 4f + 1$, this can be achieved for $\lambda \geq \log(2(2l + 2f + 3))$ resp. $\lambda \geq \log(2\lceil l/B \rceil + 2f + 3)$. The stabilization time is at most $2 \cdot 2^\lambda$ rounds.*

We remark that the dependency on l could be further decreased by the pipelining approach suggested in Section III. With respect the dependency on f, we note that there is a fundamental lower bound of $f + 1$ rounds for deterministic consensus solutions [24], which entails that $\lambda > \log(f + 1)$. Since f is at most $O(n)$, $\lambda = O(\log n)$ is hence sufficient for our solution to work. In sharp contrast to the existing self-stabilizing Byzantine fault-tolerant clock synchronization solutions surveyed in Section II, the resulting stabilization time is hence at most $O(n)$ as well. In Section VI, we will utilize randomization to further decrease the dependency on n.

V. Implementation and Experiments

In this section, we present the cornerstones of our VHDL prototype implementation and its experimental evaluation in

an FPGA, which demonstrates that our algorithms are not just of theoretical interest. Note that it is not the purpose of this implementation to optimize performance, area, or power efficiency, however.

Inspecting Algorithm 1, Algorithm 2, and Algorithm 3 reveals the need for implementing the following basic building blocks:

(i) Lock-step round simulation and synchronous state machine for executing the algorithm at each node.
(ii) Subystem for broadcasting 1-bit messages.
(iii) Threshold modules with n inputs.

For the randomized Algorithm 4, a random generator implementing coin flipping with head/tail probability $1/n$ vs. $1-1/n$ must be added.

Lock-step Round Simulation. Since assuming perfectly synchronized short clocks is unrealistic in practice, we rather assume λ'-bit wide short clocks that can differ by at most by $\pi > 0$ ticks at any point in time. We employ a standard lock-step round simulation technique for simulating a perfectly synchronized λ-bit clock with $\lambda = \lambda' - \lceil \log(2\pi + 1) \rceil$, which works as follows: Computations for consecutive lock-step rounds r and $r + 1$ need to be separated by at least π ticks, to ensure that every node's round r message arrives before a node performs round $r + 1$. This is, however, not sufficient, since a node needs to be able to determine whether a received message belongs to round r or $r+1$. We follow the simple approach of separating the computing steps by another π ticks: Node i sends the previously computed message when $C_i \bmod (2\pi + 1) = 0$ and performs computations when $C_i \bmod (2\pi + 1) = \pi + 1$ (the additional tick accounts for the setup and hold times of the message buffers, which are just flip-flops due to our single-bit channels here).

While this decreases the number of simulated rounds (and hence the effective clock frequency of the simulated short clocks) by a factor close to 2, it avoids the need for non-trivial encoding and buffering of more than a single message. Note carefully, though, that we do not a priori restrict that resolution (i.e., the clock frequency) of the underlying λ'-wide short clock. Moreover, it is possible to devise faster lock-step round simulations.

Communication subsystem. The communication system used in our implementation is very simple: At the sender side, it consists of a flip-flop that is set to the single-bit value to be communicated to all nodes in round k, according to the state machine's state, at short clock tick $k(2\pi + 1)$. The output of this sender flip-flop is routed to all receivers, that is, to the data input of a dedicated receiver flip-flop corresponding to the respective sender. It samples the data value when clock tick $k(2\pi + 1) + \pi + 1$ occurs at the receiver short clock.

Threshold modules. A threshold module with n inputs and threshold $f + 1$ resp. $n - f$ sets its single output to 1 when at least $f + 1$ resp. $n - f$ of the inputs are 1. In our implementation, we use an adder tree plus one comparator per threshold acting on the sum for this purpose. Since (i) the inputs of all the required threshold modules originate in the receiver flip-flops, and (ii) the threshold modules' output is only evaluated synchronously, i.e., when it is stable (at least for non-faulty nodes), as argued above, glitches possibly produced by this implementation cannot cause harm.

Metastability issues. Despite the guarantees provided by our lock-step round simulation, we cannot assume that setup and hold times will *never* be violated. After all, they rest upon the assumption that (a) the underlying clocks are indeed synchronized with imprecision π, which may be violated during stabilization of the short clocks, and (b) that sender and receiver are correct; Byzantine faulty nodes may not be synchronized with the other nodes, however. Hence, metastability [25] cannot be avoided in all circumstances. Nevertheless, it can reasonably be assumed that metastability does not compromise our system, for the following reasons:

(I) Guaranteed *metastability-freedom by construction in fault-free executions*, i.e., during normal operation, by our lock-step round simulation.

(II) *Low probability of metastable upsets:* The window of vulnerability of a setup/hold violation of a flip-flop is very small. In addition, mechanisms for decreasing the upset probability like synchronizers [26] or elastic pipelines acting as metastability filters [27] can easily be incorporated in our single-bit channels. Note that this increases communication latency, but does not require to further decrease clock frequency.

(III) *Metastability containment:* Non-faulty nodes are very robust w.r.t. propagation of metastable upsets, since the threshold gates usually (albeit not in all cases, and not with all implementations) mask metastable upsets in the buffers.

(IV) *Limited impact of metastable upsets:* As long as the fault-tolerance limit f is not reached, from the system's perspective upsets are masked as faults. Moreover, upsets that occur in during the stabilization phase may only delay stabilization. Since they are rare events even then, they have a very small impact on the (expected) stabilization time.

Experiments. We implemented a prototype system of $n = 8$ nodes, which tolerates at most $2 = \lfloor (8 - 1)/3 \rfloor$ persistent Byzantine faults. To facilitate systematic experiments, we also built a test bench that provides the following functionality:

(1) Start the algorithm from arbitrary (deterministic or random) states, including buffers.
(2) Reset a node to its initial state, at any time.

Setting initial states is realized by adding a scan-chain to the implementation, which allows to serially shift-in arbitrary initial system states at run-time. Repeated random experiments are controlled via a Python script executed at a PC workstation, which is connected via RS232 to an ATMega 1280 microcontroller (uC) that acts as a scan-controller towards the FPGA. The uC takes a bit-stream representing an initial configuration, sends it to the FPGA via the serial scan-chain interface, and signals the FPGA to start execution of the algorithm. When the system has stabilized, which is recognized by the monitoring unit in the testbench and signaled to the uC via a dedicated output signal, the uC informs the Python script that records the stabilization time and proceeds with sending the next initial configuration.

Consensus	faults f	1	2	∞
Phase King	2	0.737	0.263	0
Phase King	late j.	0	1.0	0
none	2	0.586	0.073	0.341

TABLE I
REL. FREQUENCY OF STABILIZATION TIMES OF ALGORITHM 1 (OVER 50000 RANDOMLY INITIALIZED RUNS).

Our test bench also provided the short clocks for all nodes. Alternatively, short clocks provided by our current FATAL$^+$ prototype implementation could be used. Moreover, the test bench also allows to monitor the internal state of all nodes and, hence, the generated round labels $L(i)$, and facilitates experiments with up to $f = 2$ Byzantine nodes. To this end, the state machines of f nodes can be replaced by special variants, which communicate conflicting information to the receivers (to keep the system as inconsistent as possible).

The test bench used for the experiments described below provides $n = 8$ nodes, up to $f = 2$ of which may be Byzantine faulty, with a synchronized $\lambda' = 9$-bit short clock, which is incremented every 40 ns. Its least significant bit hence corresponds to a 12.5 MHz periodic signal, which is further divided by 2 in order to obtain the clock signal driving the state machine and the send-flip-flops (rising transition) and receive-flip-flops (falling transition). This simplification entails that a simulated round takes 4 clock ticks and the simulated perfect short clocks have $\lambda = 7$ bits. Note that we choose this low clock frequency just for (a) convenience and simplicity of experiment control and (b) to match the clock frequency provided by the FATAL$^+$ prototype implementation. Without these constraints, it would of course be possible to dramatically raise the clock frequency even in our FPGA implementation.

The entire implementation has been written in VHDL and compiled for an Altera Cyclone IV FPGA using the Quartus tool. For $l = 16$-bit labels and $n = 8$, it consumes 3407 (resp. 4193 for late joiners) logic blocks, about 375 per node and 407 (resp. 1193) for the test bench.

Results. The first row in Table I shows the stabilization times (measured in wrap-arounds of the short clock) and their relative frequency provided by Algorithm 1 with the Phase King protocol, for $f = 2$ Byzantine nodes. In each run, all nodes start simultaneously from random initial states, at random times of the short clock. A stabilization time of 1 means that the nodes reached agreement on the round labels already at the very first wrap-around after their initialization; stabilization time 2 corresponds to runs that stabilized after the first complete execution of the while loop. As predicted by the analysis, all nodes are in synchrony at the 2^{nd} wrap-around. The second row gives the stabilization time for the case where some node joins an already running system. Late joining almost always completed after the first complete execution of the while loop of the joining node, as expected. The results for the fault-free case $f = 0$ are very similar and hence omitted.

In every run in the above experiments, we also determined whether the reduction (Algorithm 2) alone, i.e., without the subsequent Phase King, would have been sufficient to reach consensus on $L(i)$. (Note that this is always true in the special case where *all* communication is by broadcasts, i.e., even faulty nodes send the same messages to all recipients.) The third row in Table I shows the stabilization times for this setting, where the clock is always set to the value c_i returned by Algorithm 2. It is apparent that a significant number of runs never stabilize, as our Byzantine nodes persistently disseminate inconsistent information; as they follow a deterministic pattern, there are no runs that stabilize after the first complete iteration of the while loop of the algorithm. Even though such worst-case behavior is not very likely in practice, this reveals that omitting the consensus protocol is usually not an option.

VI. A RANDOMIZED ALGORITHM

In this section, we present a randomized solution for the (λ, l)-round labeling problem, based on merging a novel randomized self-stabilizing Byzantine fault-tolerant algorithm into Algorithm 1. It uses only 1-bit channels and and achieves binary consensus among $n \geq 3f+1$ nodes in $2n$ rounds with probability $1-2^{\Omega(n)}$. Randomization in conjunction with local counters is used to overcome the need for large synchronized clocks required by deterministic consensus algorithms, such that a short clock with only 3 bits is sufficient to execute the consensus algorithm. We assume that faulty nodes cannot predict random choices before they are made; apart from this, the adversary has full knowledge of the system, including communication between non-faulty nodes.

For conciseness, we will present and analyze this algorithm in conjunction with Algorithm 1. The combined algorithm is shown in Algorithm 4. For notational simplicity, we identify ***true*** $= 1$ and ***false*** $= 0$.

As in Section III and Section IV, we assume that the short clocks are perfectly synchronized ($\pi = 0$) and that an iteration of the while-loop is started at node i whenever its short clock $C_i(t)$ reads 0. Informally, the consensus part of Algorithm 4, consisting of the outermost for-loop, uses l *voting phases* (termed phases for conciseness) to reach agreement on the binary value b_i, initially supplied by Algorithm 2. In each phase, it first determines whether there is already an overwhelming number ($\geq n - f$) of nodes with the same $b_i = b$. If so, and if this fact is recognized by $\geq n - f$ nodes, the variable locked is set to ***true***. If this ever happens simultaneously at all nodes, all b_i (at correct nodes) will be identical in all subsequent phases.

The purpose of the code starting at Line 13 is to eventually achieve this. The value $b_i = b'$ of one (or more) processor(s) determined by the random choice in Line 16 will force $b_i = b'$ at all correct processors in Line 22; the $(f + 1)$-threshold in Line 9 ensures that no correct node will ever try to convince others to set $b_i = 0$ if there is a node with $b_i = 1$ and locked $=$ ***true*** (or vice versa). The counters $\Delta_{i,j}$ limit the ability of Byzantine faulty nodes to thwart this process repeatedly, as their information is only considered once in every n consecutive voting phases.

Definition VI.1 (Voting Phases)**.** *An iteration of the outer for-loop within the while-loop of Algorithm 4 is called a* voting

Algorithm 4: Randomized (λ, l)-round labeling algorithm at node $i \in V$. The persistent variables $\Delta_{i,j}$ can take values from the range $1, \ldots, n$.

```
1  while true do
2      (c_i, b_i) := red(L(i)) // reduction
3      for l times do
4          broadcast b_i
5          if received ≥ n − f times b then
6              broadcast 1b // b is unique
7          else broadcast 00 // 2 rounds for 2-bit bcast
8          locked := false
9          if received ≥ f + 1 times 1b then
10             b_i := b // b remains unique
11             if received ≥ n − f times 1b then locked := true
12         for j ∈ V do Δ_{i,j} := max{0, Δ_{i,j} − 1}
13         for b ∈ {0, 1} do
14             p_b := false // indicates if b is proposed
15             if Δ_{i,i} = 0 and b_i = b then
16                 broadcast 1 with probability 1/n
17             else broadcast 0
18             for each node j that sent 1 do
19                 if Δ_{i,j} = 0 then p_b := true
20                 Δ_{i,j} := n
21             broadcast p_b
22             if locked = false and received ≥ n − f times true
               then
23                 b_i := b
24                 locked := true
25     if b_i = 1 then L(i) := c_i
26     else L(i) := 0
27     wait until round number modulo 2^λ is 0
28     L(i) := L(i) + 1 mod 2^l // increase clock
```

phase.

We start our proof by showing that a safe configuration will be maintained deterministically.

Lemma VI.2. *If after Line 2 or any phase of Algorithm 4 all non-faulty nodes i have the same values b_i and c_i, this statement becomes an invariant. Moreover, $c_i = L(i)$ after Line 2 of subsequent phases and $L(i)$ will be increased by $1 \bmod 2^l$.*

Proof: For the first claim of the lemma, we need to show (i) that the invariant does not become violated when executing a phase at whose beginning it holds and (ii) that it does not become violated when executing Line 2 if it held after the last phase of the previous iteration of the while-loop.

To see (i), observe that if $b_i = b$ for all (at least $n - f$) non-faulty i, they will all broadcast $1b$, "set" $b_i = b$ and locked = ***true***, and therefore not modify b_i during the phase. Since during a phase c_i and $L(i)$ cannot be changed, this shows (i).

To see (ii), observe that after the last voting phase of the previous while-loop, all $L(i)$ are set to a unique value $L = c_i$ (at non-faulty nodes i). Subsequently, each such node sets $L(i) := L(i) + 1 \bmod 2^l$. Hence, by Theorem III.2, the call to Algorithm 2 in Line 2 will return $(L+1 \bmod 2^l, b)$ (with b being unanominously either ***true*** or ***false***). This proves (ii) and also shows that $L(i)$ will increase by $1 \bmod 2^l$ in subsequent iterations of the while-loop, completing the proof. ■

A simple condition for stabilization follows.

Corollary VI.3. *If in any phase of a complete iteration of its while-loop all non-faulty nodes i have locked = **true** and $b_i = b$, Algorithm 4 has stabilized once the current iteration of the while-loop is complete.*

Proof: Examining the code, we see that the precondition of the corollary becomes satisfied in some round after the execution of Line 8 of the respective phase. Since all nodes have locked = ***true***, in the latter case they will not change their values b_i. By Theorem III.2, after Line 2 at the beginning of the current iteration non-faulty nodes share the same value c_i. Therefore, the precondition of Lemma VI.2 becomes satisfied at the end of the phase in which the precondition of the corollary becomes true, and the claim of the corollary follows from the lemma. ■

Our goal is to ensure that Corollary VI.3 can eventually be applied. To this end, we will use the following definition.

Definition VI.4 (Active Phases)**.** *Non-faulty node i is* active *in a given phase if it satisfies that $\Delta_{j,i} = 0$ for all non-faulty nodes j at its beginning. A phase is* active *if at least $n/12$ nodes are active in this phase.*

We next establish that within linear time, with overwhelming probability there are many active phases.

Lemma VI.5. *For any $m > n$, the m^{th} voting phase is active with probability $1 - 2^{-\Omega(n)}$. With probability $1 - 2^{-\Omega(n)}$ all phases $n+1, \ldots, 2n$ are active.*

Proof: In voting phase m, each non-faulty node i will satisfy that $\Delta_{j,i} = 0$ for all non-faulty nodes j unless i broadcasted 1 in Line 16 in one of the voting phases $m - n, \ldots, m - 1$ (since $\Delta_{j,i}$ decreases by one in each such phase without such a broadcast). Due to independence of the random choices, the probability i is active in round m is thus at least $(1 - 1/n)^n \approx 1/e$, independently for each i. Since there are at least $n - f > 2n/3$ non-faulty nodes, the expected number of active nodes in phase m is at least $2n(1 - 1/n)^n/3 \approx 2n/(3e)$. By Chernoff's bound, it follows that the probability that less than $n/5$ nodes are active is lower bounded by $1 - 2^{-\Omega(n)}$. The union bound then yields second statement of the lemma. ■

Definition VI.6 characterizes phases in which the algorithm is certain to reach agreement. It is used in a few technical lemmas, which establish Theorem VI.10.

Definition VI.6 (Good Phases)**.** *A phase is called* good *if (i) an active node broadcasts 1 in the first execution (for $b = 0$) of Line 16 of that phase or (ii) at most f non-faulty nodes broadcast $p_0 =$ **true** in the first execution of Line 21 of the phase and an active node broadcasts 1 in the second execution of Line 16 (for $b = 1$) of that phase.*

Lemma VI.7. *In a good phase of a complete execution of the while-loop all non-faulty nodes i have locked = **true** and the*

same value b_i at some point.

Proof: **Case 1:** An active node broadcasts 1 in the first execution of Line 16 of the phase. Hence, it received at most f times 11 in Line 9, implying that no non-faulty node i received more than $2f < n - f$ times 11 and set locked := ***true*** and $b_i = 1$. Therefore, all non-faulty nodes with locked = ***false*** set locked := ***true*** and those with $b_i = 1$ set $b_i := 0$ in Line 24.
Case 2: An active node broadcasts 1 in the second execution of Line 16 of the phase. By definition of good phases, at most f non-faulty nodes broadcast $p_0 =$ ***true*** in the first execution of Line 21 of the phase. Thus, no non-faulty nodes i sets $b_i := 0$ or locked := ***true*** in the first execution of the second for-loop of the phase. Thus, we can argue analogously to Case 1: no non-faulty node i has locked = ***true*** and $b_i = 0$ at the beginning of the second iteration of the second for-loop of the phase, and therefore all nodes will have $b_i = 1$ and locked = ***true*** at its end. ■

We need to show that despite the Byzantine nodes' interference, a good phase occurs within n phases.

Lemma VI.8. *Each active phase $m > n$ is good with constant probability or at least $f + 1$ variables $\Delta_{j,k}$, j non-faulty and k faulty, are reset to n. This holds independently of random choices made in other phases and the state of the nodes at the beginning of the phase.*

Proof: **Case 1:** At least half of the active nodes, i.e., $n/24$, have $b_i = 0$ after Line 11 of the phase. With probability at least $1 - (1 - 1/n)^{n/24} \in \Omega(1)$ at least one them broadcasts 1 in the first execution of Line 16 of that phase. The claim follows by Lemma VI.7.
Case 2: At least $n/24$ active nodes have $b_i = 1$ after Line 11 of the phase.
Case 2a: A non-faulty node i broadcasts 1 in the first execution of Line 16. Since we consider phase $m > n$, such a node has not broadcasted in Line 16 for the previous $n - 1$ phases: otherwise it would have set $\Delta_{i,i} = n$ and not decreased it by more than $n - 1$ until the current phase. Hence, the node is active and the claim follows analogously to Case 1.
Case 2b: At least $f + 1$ variables $\Delta_{j,k}$, j non-faulty and k faulty, are reset to n in the first iteration of the second for-loop of the phase. The claim is immediate.
Case 2c: No non-faulty node i broadcasts 1 in the first execution of Line 16 and at most f variables $\Delta_{j,k}$, j non-faulty and k faulty, are reset to n in the first iteration of the second for-loop of the phase. Since non-faulty nodes broadcast 0 in Line 16, each non-faulty node j that broadcasts $p_0 =$ ***true*** in the first execution of Line 21 must reset $\Delta_{j,k} := n$ for some faulty node k. Since there are at most f such events, non-faulty nodes i receive at most $2f < n - f$ times ***true*** in Line 21 and do neither modify locked nor b_i. Hence, there are still at least $n/24$ active nodes with $b_i = 1$ when Line 16 is executed for the second time in this phase. With probability at least $1 - (1 - 1/n)^{n/24} \in \Omega(1)$, at least one of them will broadcast 1, implying that the phase is good. Because the cases are exhaustive, this concludes the proof. ■

Lemma VI.9. *With probability $1 - 2^{-\Omega(n)}$, one of the phases $n + 1, \dots, 2n$ is good.*

Proof: By Lemma VI.5, all phases $n + 1, \dots, 2n$ are active with probability $1 - 2^{-\Omega(n)}$. Conditioning on this event, we apply Lemma VI.8 to each of these phases, showing that each of them is good with independently and constantly lower bounded probability unless at least $f + 1$ variables $\Delta_{j,k}$ for non-faulty nodes j and faulty nodes k are set to n in the respective phase. As it takes at least n phases for these variables to decrease to 0 again, during phases $n + 1$ to $2n$ this can happen at most once for each such variable. There are $f(n - f)$ such variables, implying that there can be at most $f(n - f)/(f + 1) < n - f < 2n/3$ phases in which more than f of them are set from 0 to n. Hence at least $n/3$ phases remain that are good with independent and constant probability.

By Chernoff's bound, with probability $1 - 2^{-\Omega(n)}$ one of these phases is good. Thus, the unconditional probability that a good phase occurs is $(1 - 2^{-\Omega(n)})^2 = 1 - 2^{-\Omega(n)}$. ■

Theorem VI.10. *Suppose that $\lambda \geq \lceil \log(9l + 1) \rceil$. Then, with 1-bit channels, Algorithm 4 solves (λ, l)-round labeling with probability 1, and actually stabilizes in $\mathcal{O}(l + n)$ rounds with probability $1 - 2^{-\Omega(l+n)}$.*

Proof: It is easy to see that each iteration of the while-loop of Algorithm 4 takes $9l + 1$ rounds, $2l + 1$ for the call to Algorithm 2 (by Theorem III.2) and $7l$ since each voting phase takes 7 rounds. Hence, if the wrap-around of the common clock occurs every 2^λ rounds, this is sufficient to control the execution of the while-loop. By Lemma VI.2, the algorithm will have stabilized once an iteration of the loop is complete in which the preconditions of the lemma are met in some round. According to Lemma VI.7, this will happen once a phase is good, which by Lemma VI.9 is guaranteed to be the case for some phase $m \in \{n + 1, \dots, 2n\}$ with probability $1 - 2^{-\Omega(n+l)}$. Since we make no assumptions on the initial state of the variables of the algorithm, this result can actually be applied to any sequence of $2n$ phases. We conclude that the algorithm stabilizes with probability $1 - 2^{-\Omega(l+n)}$ once (i) the first (not necessarily complete) iteration of the while-loop is complete and n phases have passed and (ii) another n phases have passed and the iteration of while-loop containing the last of these phases is complete. This takes $\mathcal{O}(n+l)$ rounds, proving the first part of the claim. The second follows by recalling that the first statement applies independently of the inital state and applying it repeatedly. ■

We remark that using B-bit channels, pipelining, and reducing the number of voting phases per iteration of the while-loop can all reduce the minimal feasible value of λ. In the event that λ is still too small, we can apply the technique recursively, at small overhead either in terms of bandwidth or running time.

Definition VI.11 (Log-star). *For $x \in \mathbb{R}^+$, we define that $\log^* x := 0$ if $x \leq 1$ and $\log^* x := 1 + \log^* \log x$ else.*

Corollary VI.12. *Suppose that $\lambda \in \mathcal{O}(1)$ is sufficiently large*

and nodes can broadcast $B \in \mathcal{O}(\log^* l)$ bits in each round. Then (λ, l)-round labeling can be solved with probability 1, stabilizing in $\mathcal{O}(l+n)$ rounds with probability $1-2^{-\Omega(l+n)}$.

Proof: Consider the algorithm where we run B copies of Algorithm 4 on top of each other, each using the clock the subjacent instance outputs to control its while-loop. The communication is done in parallel by using one bit in each round for each instance.

The first instance needs to rely on the the given clock with wrap-around every $2^\lambda \in \mathcal{O}(1)$ rounds in order to control its while-loop. Maximizing the value l_1, the number of clock bits in its output, yields an exponentially larger clock of $\lambda_2 := l_1 \in 2^{\Omega(\lambda)}$ bits. Now we have a clock of λ_2 bits that we can utilize in the second instance to produce an even larger clock for the third instance. Proceeding inductively using the relations $\lambda_{i+1} := l_i$ and $l_i \in 2^{\Omega(\lambda_i)}$ (for $i \in \{1, \dots, B-1\}$), we can see that the number of clock bits produced by the top instance is an iterated exponential in B. Hence, for $B \in \mathcal{O}(\log^* l)$, we will arrive at clocks of $l_B \geq l$ bits as desired.

By Theorem VI.10, the i^{th} instance of the algorithm will stabilize within $\mathcal{O}(n+l)$ rounds with probability $1-2^{-\Omega(n+l)}$ once the $(i-1)^{th}$ instance has done so (except for $i=1$, where this statement is unconditional), and by the union bound this will happen for all instances with probability $1-2^{-\Omega(n+l)}\log^* l = 1-2^{-\Omega(n+l)}$. However, following this naive approach, the total time to stabilize all instances with this probability is $\mathcal{O}((n+l)\log^* l)$.

A minor modification of the algorithm can get rid of this factor $\log^* l$ overhead. Instead of handling the values $\Delta_{j,k}$, $j,k \in V$ for each instance separately, we now maintain only one set of these variables. $\Delta_{j,k}$ is set to n if node k broadcasts a 1 in Line 16 for *any* of the concurrently running instances of the algorithm. The decision whether to broadcast in Line 16 of the algorithm is also made for all instances (that currently are in a voting phase) together. That is, if $\Delta_{j,j} = 0$ and in any instance node j satisfies that and $b_j = b$ in the condition to execute Line 16, the node will do so with probability $1/n$ for all instances satisfying $b_j = b$. Note that this causes no problems if we align the voting phases of all instances that are currently not executing Algorithm 2, which does not affect the asymptotic growth of the size of clocks.

This modification implies that the new algorithm satisfies that (i) each node executes Line 16 with probability at most $1/n$ in a given round, (ii) each node that is active in any instance executes the line with probability $1/n$ in the respective round, and (iii) the number of variables $\Delta_{j,k}$ with j non-faulty and k faulty is at most $f(n-f)$. Hence, the same reasoning as in Theorem VI.10 applies again, however, as the variables $\Delta_{j,k}$ are now the same for all instances, the faulty nodes now cannot stall all instances for a linear number of rounds.

Formally, we adapt the definitions of active nodes and phases and good phases so that they apply to the instance of the algorithm that is lowest in the hierarchy that has not yet stabilized. Applying Chernoff's bound in essentially the same manner as in Lemma VI.9, we can in fact infer that there are with probability $1-2^{-\Omega(n+l)}$ at least $\Omega(n+l) \supset \Omega(\log^* l)$ good phases in which there is a round satisfying the prerequisites of Lemma VI.2 for the next instance that has not stabilized yet.[3] We conclude that for all instances $i \leq i_0$, where i_0 is such $l_{i_0} \geq n$ (if there such an i_0, otherwise $i_0 = B$), the probability to stabilize within $\mathcal{O}(n)$ rounds is at least $1-2^{-\Omega(n)}$. For the remaining levels $i \geq i_0$, l_i is the dominating term in the time complexity. Overall, we infer the statement of the corollary. ∎

Corollary VI.13. *Suppose that $\lambda \in \mathcal{O}(1)$ is sufficiently large. Then (λ, l)-round labeling can be solved with probability 1 using 1-bit channels and stabilization in $\mathcal{O}((l+n)\log^* l)$ rounds with probability $1-2^{-\Omega(l+n)}$.*

Proof: We modify the algorithm from the previous corollary to operate with 1-bit channels. To this end, we use time division to simulate larger bandwidth, trading in a reduction in bandwidth requirements for a larger time complexity. Note that if we simulate a bandwidth that is by factor f larger, this necessitates also clocks that have a by factor f larger wrap-around. Clearly this is not a problem once the clock wrap around happens after $\Omega(\log^* l)$ rounds (for sufficiently large constants). In order to resolve this bootstrapping issue, we reserve, say, a fraction of $1/3$ of all rounds solely for one instance of the algorithm from the previous corollary. With the resulting larger clocks, we can run (at least) three instances concurrently (using time division) in a fraction of $1/9$ of the rounds reserved solely for this purpose. Repeating this process and taking into account that the number of instances that we can run concurrently using time division grows exponentially in each step, we can bound the fraction of rounds occupied so far by the geometric series in $1/3$, i.e., we will reserve a fraction of $1/2$ of the rounds that we can still use.

We halt this process once we arrive at sufficiently large clocks, i.e., a wrap-around of $\Omega(\log^* l)$. Applying Corollary VI.12 to each instance, we see that the total time required for stabilization with probability $1-2^{-\Omega(n+l)}$ is $\mathcal{O}(n+l)$ times the inverse fraction of the assigned rounds (accounting for the increase in running time due to time division). Thus, the total stabilization time for this probability threshold, which by the union bound is upper bounded by the sum of the individual stabilization times, is asymptotically dominated by the stabilization time of either the final or the second last instance. As we reserved a constant fraction of the round for the final instance and it simulates bandwidth $\mathcal{O}(\log^* l)$ by means of time division, it stabilizes within $\mathcal{O}((n+l)\log^* l)$ rounds with probability $1-2^{-\Omega(n+l)}$. The second last instance needs to construct clocks of $\mathcal{O}(\log^* l)$ bits with access to clocks of size $\mathcal{O}(\log^* \log^* l)$, and it is assigned a fraction of at least $2^{-\mathcal{O}(\log^* \log^* l)}$ of the rounds. Since $2^{\mathcal{O}(\log^* \log^* l)} \log^* \log^* l \subset \mathcal{O}(\log^* l)$, the stabilization

[3] A detail is here that there may be part of an exectution of the while-loop of the algorithm that is "wasted time" because the nodes did not agree on the current clock value until the next lower level stabilized. Note, however, that even when summing over all instances this amounts to a total of $\mathcal{O}(l)$ "lost" rounds.

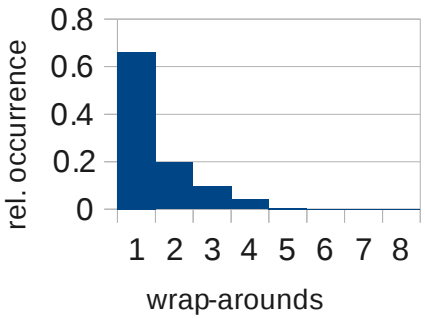

Fig. 1. Stabilization times of Algorithm 4 (over 50000 randomly initialized runs with $f = 2$).

time of the last instance is asymptotically dominant, proving the corollary. ∎

Experiments. We conclude this section with a brief glance of the performance of our implementation of Algorithm 4. For $l = 4$-bit labels and $n = 8$, it consumed 4597 logic blocks, with about 545 per node, in the setup described in Section V. Figure 1 shows the stabilization times (measured in wrap-arounds of the short clocks) for $f = 2$ Byzantine faulty nodes. It can be seen that the algorithm stabilizes within only 2 wrap-arounds in most of the cases.

VII. Conclusion

For arbitrary integer l, we demonstrated how to efficiently establish $(\lambda+l)$-bit-wide synchronized clocks at each of the n nodes of a SoC, provided every node is already equipped with a λ-bit-wide synchronized short clock. Our generic Byzantine fault-tolerant self-stabilizing solution uses a reduction to Byzantine fault-tolerant binary consensus and requires only a few bit wide channels between nodes. Different solutions can be obtained by plugging in different consensus implementations: Utilizing the deterministic phase king algorithm, a solution for $\lambda \in \mathcal{O}(\log n)$ that stabilizes in $\mathcal{O}(n)$ time is obtained; utilizing a novel randomized consensus algorithm even provides a solution for $\lambda \in \mathcal{O}(1)$ that stabilizes in $\mathcal{O}(n + l)$ time with probability 1. FPGA implementations of our solutions, which demonstrate that they can indeed be implemented directly in VHDL, have been used for experimentally validation of the results of our correctness proofs and analytic performance analyses.

References

[1] L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 254–280, Apr 1984.

[2] A. Hansson, M. Subburaman, and K. G. W. Goossens, "Aelite: A Flit-Synchronous Network on Chip with Composable and Predictable Services," in *Proceedings Design, Automation & Test in Europe Conference and Exhibition (DATE 2009), Nice, France*. Los Alamitos: IEEE Computer Society, April 2009, pp. 250–255.

[3] R. Obermaisser, H. Kopetz, C. El Salloum, and B. Huber, "Error Containment in the Time-Triggered System-On-a-Chip Architecture," in *Embedded System Design: Topics, Techniques and Trends*, ser. IFIP, A. Rettberg, M. Zanella, R. Dömer, A. Gerstlauer, and F. J. Rammig, Eds. Springer US, 2007, vol. 231, pp. 339–352.

[4] R. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept. 2005.

[5] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, pp. 228–234, 1980.

[6] P. Berman, J. A. Garay, and K. J. Perry, "Asymptotically Optimal Distributed Consensus," 1992. [Online]. Available: http://www.bell-labs.com/user/garay/#distributed-pub

[7] T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, 1987.

[8] J. L. Welch and N. A. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, vol. 77, no. 1, pp. 1–36, 1988.

[9] P. Thambidurai, A. Finn, R. Kieckhafer, and C. Walter, "Clock Synchronization in MAFT," in *19th Int. Symp. on Fault-Tolerant Computing*, Jun 1989, pp. 142 –149.

[10] P. S. Miner, "Verification of Fault-Tolerant Clock Synchronization Systems," *NASA Technical Paper 3349*, Nov. 1993.

[11] W. Steiner and M. Paulitsch, "The Transition from Asynchronous to Synchronous System Operation: An Approach for Distributed Fault-Tolerant Systems," *Proceedings of the The 22nd International Conference on Distributed Computing Systems*, Jul 2002.

[12] J. Widder and U. Schmid, "Booting Clock Synchronization in Partially Synchronous Systems with Hybrid Process and Link Failures," *Distributed Computing*, vol. 20, no. 2, pp. 115–140, Aug 2007.

[13] D. Dolev and E. Hoch, "Byzantine Self-Stabilizing Pulse in a Bounded-Delay Model," in *Proc. 9th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, vol. 4280, 2007, pp. 350–362.

[14] S. Dolev and J. L. Welch, "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults," *Journal of the ACM*, vol. 51, no. 5, pp. 780–799, 2004.

[15] A. Daliot, D. Dolev, and H. Parnas, "Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks," in *Proc. 6th Symposium on Self-Stabilizing Systems (SSS)*, 2003.

[16] A. Daliot and D. Dolev, "Self-Stabilizing Byzantine Pulse Synchronization," *Computing Research Repository*, vol. abs/cs/0608092, 2006.

[17] M. Malekpour, "A Byzantine-Fault Tolerant Self-stabilizing Protocol for Distributed Clock Synchronization Systems," in *Proc. 9th Conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2006, pp. 411–427.

[18] ——, "A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol," NASA, Tech. Rep., 2009, tM-2009-215758.

[19] D. Dolev, M. Függer, C. Lenzen, and U. Schmid, "Fault-Tolerant Algorithms for Tick-Generation in Asynchronous Logic: Robust Pulse Generation - [Extended Abstract]," in *Proc. 13th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2011, pp. 163–177.

[20] D. Dolev, M. Függer, C. Lenzen, and U. Schmid, "Fault-Tolerant Algorithms for Tick-Generation in Asynchronous Logic: Robust Pulse Generation," *Computing Research Repository*, vol. abs/1105.4780, 2011.

[21] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep 2002. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[22] B. Kemme, A. Bartoli, and O. Babaoglu, "Online Reconfiguration in Replicated Databases Based on Group Communication," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, Jul 2001, pp. 117 –126.

[23] P. Berman, J. A. Garay, and K. J. Perry, *Bit Optimal Distributed Consensus*. New York, NY, USA: Plenum Press, 1992, pp. 313–321.

[24] D. Dolev and H. R. Strong, "Polynomial Algorithms for Multiple Processor Agreement," in *Proc. 14th Symposium on Theory of Computing (STOC)*, 1982, pp. 401–407.

[25] L. Marino, "General Theory of Metastable Operation," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 107–115, 1981.

[26] D. J. Kinniment, A. Bystrov, and A. V. Yakovlev, "Synchronization Circuit Performance," *IEEE Journal of Solid-State Circuits*, vol. SC-37, no. 2, pp. 202–209, 2002.

[27] G. Fuchs, M. Függer, and A. Steininger, "On the Threat of Metastability in an Asynchronous Fault-Tolerant Clock Generation Scheme," in *Proc. 15th Symposium on Asynchronous Circuits and Systems (ASYNC)*, Chapel Hill, N. Carolina, USA, 2009, pp. 127–136.