DISTRIBUTED COMPUTING COLUMN

Stefan Schmid
TU Berlin & T-Labs, Germany
stefan.schmid@tu-berlin.de

Fault-tolerant Distributed Systems in Hardware

Danny Dolev (Hebrew University of Jerusalem)
Matthias Függer (MPI for Informatics)
Christoph Lenzen (MPI for Informatics)
Ulrich Schmid (TU Vienna)
Andreas Steininger (TU Vienna)

Very large-scale integrated (VLSI) hardware designs can be seen as distributed systems at several levels of abstraction: from the cores in a multicore architecture down to the Boolean gates in its circuit implementation, hardware designs comprise of interacting computing nodes with non-negligible communication delays. The resulting similarities to classic large-scale distributed systems become even more accented in mission critical hardware designs that are required to operate correctly in the presence of component failures.

We advocate to act on this observation and treat fault-tolerant hardware design as the task of devising suitable distributed algorithms. By means of problems related to clock generation and distribution, we show that (i) design and analysis techniques from distributed computing can provide new and provably correct mission critical hardware solutions and (ii) studying such systems reveals many interesting and challenging open problems for distributed computing.

1 Introduction

Very large-scale integrated (VLSI) circuits bear several similarities with the systems studied by the distributed computing community:

- They are formed by an increasing number of interacting computing nodes.
- Communication delays are not negligible.
- The cost of communication, such as area and power consumption, is not negligible.

In fact, this view is correct at different levels of abstraction. We will elaborate on two such levels that significantly differ from each other with respect to the computational power of the system's nodes, their means of communication, and the problems they solve.

(I) Viewed from a low-level perspective, every digital circuit is a network of logic gates with delays, which continuously compute their current output state from their input history and interact via binary-valued, continuous-time signals. We stress the fact, however, that this is already a (convenient) abstraction, as real gates are electronic devices that process analog (continuous-valued) signals: A signal that is above a certain threshold voltage is considered high, otherwise low. Whereas analog signals (like a good clock signal) that swing fast from low voltages to high voltages are represented reasonably well by the resulting digital abstraction, this is obviously not the case for every signal: Just consider an analog signal that stays very close to the threshold voltage and just, e.g., due to very small noise, occasionally crosses it. It will turn out that this modeling inaccuracy causes serious problems both for synchronization and fault-tolerance, which must be considered explicitly.

Analogously to distributed computing, there are two fundamentally different ways to design digital circuits (i.e., algorithms in hardware), which correspond to synchronous and asynchronous algorithms in distributed computing.

The classic design paradigm relies on the synchronous computational model. It abstracts away the timing of gates and interconnects by considering gate outputs only at predetermined instants dictated by a central periodic clock signal. While this approach allows the designer to solely concentrate on the stable outputs of a network of gates, it relies critically on the guarantee that all signals have settled and all transients have vanished at the occurrence of the next clock transition. Inherently, such designs run at the speed of the clock period that is determined from worst-case bounds on gate and interconnect delays. Due to increasingly pronounced delay variations [52, 84] this results in highly conservative bounds and thus in considerable performance loss.

In contrast, designs that do not rely on the convenient discrete time abstraction provided by a clock signal are called *clockless* or asynchronous.[1] Such circuits must rely on different techniques to enforce some ordering between signal transitions. Suitable techniques range from aggressively timed circuits that explicitly use information on the delays along certain paths [80, 91, 97] to circuits that tolerate (certain) delay variations by means of some forms of handshaking. Prominent examples of the latter are *delay insensitive* (DI) circuits [72], speed-independent (SI) circuits and *quasi-delay insensitive* (QDI) circuits [74, 75]. While DI circuits are guaranteed to behave correctly in the presence of arbitrary gate and interconnect delay variations, SI resp. QDI circuits assume that all resp. certain signal forks in the interconnect are isochronic, i.e., have roughly equal propagation delays along all their fork teeth.

The robustness to delay variations in DI circuits comes at a high price, however: Martin [73] showed that the expressiveness of circuits that are DI at gate-level is severely limited. In fact, the only two-input gate allowed in such circuits is the C-Element, which is an AND gate for signal transitions; it produces a, say, rising transition at its output when it observed a rising transition at all its inputs. This clearly restricts the usability of DI circuits for real applications.

SI and QDI circuits, on the other hand, are Turing-complete [70]. Intuitively, the isochronic fork assumption guarantees that a gate whose output drives an isochronic fork implicitly performs a handshake with all its successor gates while just handshaking with one of its successors. A precise characterization of the conditions on the propagation delay that have to hold on certain paths in SI circuits was derived in [56].

(II) With the increasing number of computing nodes in *System-on-Chip* (SoC) and *Network-on-Chip* (NoC) architectures, problems that used to be relevant only in large-scale computer networks also become relevant within a single chip. Examples range from establishing a common time base over data communication and routing between nodes to load balancing.

In the hardware context, establishing a common time base among all nodes is of particular interest, because this sets the base for a synchronous computational model: Rather than being implemented entirely clockless, higher-level services like routing and load balancing could then also exploit synchrony properties. Unfortunately, however, the GHz clock speeds and sizes of modern VLSI circuits make it increasingly difficult to distribute a central clock signal throughout the whole circuit [43, 96]. Modern SoCs and NoCs hence typically rely on the *globally asynchronous locally synchronous* (GALS) approach [14], where different parts of a chip are clocked by different clock sources. Using inde-

[1] We will use the term "clockless" in the following, as such circuits do not always allow for arbitrarily large and unknown delays.

pendent and hence unsynchronized clock domains would give away the advantages of global synchrony and also requires non-synchronous cross-domain communication mechanisms or synchronizers [59, 58]. A promising alternative is mesochronous clocking [79] (sometimes also called *multi-synchronous* clocking [93]) as it guarantees some upper bound on the skew between clock domains. In this article, we will thus focus on discussing methods for providing a common time in GALS architectures.

Fault-tolerance. Besides an increasing number of components and non-negligible communication costs both at gate and system level, there is a further trend that advocates the application of distributed computing methods for designing VLSI chips: the increasing susceptibility to failures. Indeed, fault-tolerance has been identified as a key challenge in the International Technology Roadmap for Semiconductors [52] for years. Besides the increasing susceptibility of nanometer VLSI technology to permanent failures caused by manufacturing process variations and excessive operating conditions (supply voltage, temperature) [62], steadily decreasing feature sizes and signal voltage swings also led to dramatically increased transient failure rates [16], caused by ionizing particles hitting the junction of transistors [6], electro-magnetic cross-talk between signal wires and supply-voltage variations caused by simultaneous switching activities [78, 85].

Unfortunately, even relatively simple faults unveil the very limited ability of the convenient digital signal abstraction to properly describe reality. For example, an out-of-spec output driver of a gate that drives a fork to two different gate inputs may be able to reach the threshold voltage at one input but not at the other, causing those to interpret the gate output inconsistently. Similarly, a *single-event transient* (SET) [6] caused by an ionizing particle that hits the output driver of such a gate may be visible at one input but not at the other, depending on the latter's input capacitances. It is hence apparent that classic benign failure models from distributed computing, where a message is either lost or transmitted correctly, do not cover such faults. In fact, faulty gates have to be assumed to potentially behave arbitrarily, i.e., Byzantine [86].

While there is a huge body of work on fault mitigation techniques at the technological level (like *silicon-on-insulator* (SOI) technology [76]) and gate level (like the SET-tolerant DICE latch [83]), keeping the overall error rate acceptable [89, 22] despite the tremendously increasing number of gates/cores on a single chip demands for additional architectural solutions. At the same time, solutions that require central knowledge of the current system state (i) become infeasible due to the high communication costs and (ii) would themselves form a single point of failure. Algorithmic solutions that use only local knowledge, studied by the distributed computing community for decades, are hence promising in this

context.

Classic architectural fault-tolerance approaches [87] like *Dual Modular Redundancy* (DMR) and *Triple Modular Redundancy* (TMR) fail in absence of a global time base, as it becomes unclear over which values to vote. Jang and Martin [53] adapted this method to QDI designs and applied it to build a microcontroller tolerating certain transient faults [54], in particular, *single-event upsets* (SEUs), where a state-holding device may flip its state due to a particle hit. Their solution duplicates each gate and adds two succeeding cross-coupled C-Elements whose inputs are connected to the outputs of the duplicated gates. In case of a spurious state transition of one of the duplicated gates, both C-Elements do not propagate the spurious output value until it is matched by the other gate's output also (which can be proved to eventually happen). While this method tolerates SEUs, it neither allows to tolerate SETs nor permanent faults.

Tolerating such faults necessarily requires to extend the circuit's control logic not to wait for all of its predecessors' outputs. In contrast to the AND-causality semantics of the C-Element, this requires OR-causality semantics. Interestingly, there has been research in this direction in a different context: In certain cases, a Boolean function's value can already be determined from a subset of its parameters. This fact can be used to speed up clockless circuits [17, 95]. Instead of waiting for all of a module's inputs to arrive, the module waits until its outcome is determined and then immediately produces a new output value. Care must be taken not to mix up current input data with lately arriving input data from a previous computation, however. The approach thus requires additional timing assumptions and ways to memorize which input values a module already took into account when computing its output.

A similar strategy can also be applied to design clockless circuits that tolerate a certain fraction of its input nodes to fail permanently. In particular, it has also been employed in the DARTS Byzantine fault-tolerant clock generation approach [49, 44] for mesochronous GALS architectures, which comprises a network of interconnected OR-causality gates whose outputs generate tightly synchronized clock pulses. The approach is discussed in more detail in Section 3.1.3.

The limit of the fault-tolerant hardware solutions discussed above is that they allow only a certain subset of the components to fail. Even if these components start to operate according to their specification again later on, their state may remain corrupted, preventing them from recommencing correct operation. For massive transient failures, which are likely to occur e.g. in space missions and may corrupt the entire system state, the above solutions are not adequate. To attack this problem, the concept of self-stabilizing distributed algorithms [33] has successfully been applied to digital circuits. A self-stabilizing circuit is guaranteed to eventually behave correctly again after its state was arbitrarily corrupted. For example, a self-stabilizing token passing algorithm implemented in clockless hard-

ware was presented in [11], and S. Dolev and Haviv [34] built and proved correct a self-stabilizing microprocessor.

From a robustness point of view, a combination of resilience to permanent faults and self-stabilization is most desirable: Appropriate solutions operate correctly in the presence of not too many permanent faults and even recover from massive transient disruptions. In [27, 28] the fault-tolerant and self-stabilizing pulse generation algorithm FATAL and its hardware implementation were presented and proven correct. This solution is discussed in more detail in Section 3.1.4.

Additional challenges. While we highlighted major similarities between VLSI designs and classic distributed systems, there are also important differences. In most cases, these advise against a naive implementation of distributed algorithms in hardware.

First and foremost, this is the absence of computationally powerful atomic actions at the gate level in clockless circuits: Explicit means such as handshaking must be employed to synchronize activities in such a circuit, which is not only costly but also imperfect in terms of synchronization accuracy. This, in turn, is also a source for a unique problem called *metastability*, which arises when a circuit must handle input signals that bear no well-defined timing relation to its own state transitions. Consider a simple R/W register in a shared memory system that my be read by a processor at the time it is written by some other processor. Distributed computing models based on regular or even atomic registers assume that either the previous or the newly written value is returned by the read. In reality, the situation is even worse than assumed for safe registers, which allow an arbitrary return value in this case: The register may reach a metastable state, which moves its output voltage to a non-digital level for an unpredictable time!

Another unique problem arises from the imperfect coverage of the digital abstraction for analog signals in the case of failures. In distributed computing, Byzantine behavior is considered the worst a component can do to a system. Unfortunately, in digital circuits, generating an arbitrary binary-valued continuous-time signal is not the worst behavior of a component. Rather, a component may produce an arbitrary analog signal on its output, e.g., an output voltage that remains very close to the threshold voltage arbitrarily long, which is actually one manifestation of metastability (creeping metastability) [7, 13]. We will discuss these issues in more detail in Section 2.

Structure of this article. We start in Section 2 with a discussion on the peculiarities of SoCs in comparison to classic distributed systems, and the challenges arising in the definition of an appropriate distributed system model. In Section 3,

we discuss the problem of obtaining a common time base for multi-synchronous GALS architectures, which is both fundamental to the solution of other problems and exemplarily captures the challenges of adapting distributed algorithms for use on a chip. The problem is divided into three parts: (i) Section 3.1 discusses the problem of generating synchronous pulses. (ii) Section 3.2 deals with establishing local counters. Here we provide more technical details, with the primary goal of illustrating how techniques from distributed computing find application in VLSI design. (iii) Section 3.3 finally is concerned with distributing the clock over a wider area. The work is concluded in Section 4.

2 Modeling Issues

While we pointed out the similarities of VLSI circuits and fault-tolerant distributed systems in Section 1, a simple migration of classic solutions in distributed computing is not favorable and most of the time even infeasible. The most prominent obstacles are:

(i) Gates continuously compute their output state from their input states. They generate events, i.e., binary transitions, in a fully parallel way and are capable of very simple computations, such as the logical AND of two binary inputs, only. Any kind of event sequencing and atomic actions that group several binary transitions into more powerful computations requires explicit synchronization between the concurrently operating gates, e.g., by handshaking or timing assumptions. Note that this includes even "simple" computations such as the sum or the product.

(ii) Communication and computation is costly, especially if the proposed solution is meant to "only" provide low-level services to the application running on top. For example, clock generation algorithms must not use more than a few wires between nodes to be able to compete with classic clock distribution networks. Exchange of data, even a few bits, requires parallel or serial coding and decoding logic and thus typically cannot be afforded for low-level services. Rather, solutions must resort to signaling a few status bits only. Exchanging data of more than, say, 32 bits, is usually also difficult for high-level services.

(iii) Non-digital low-level effects must be taken into account. Every binary valued model necessarily abstracts from the analog signals in real gate implementations. While it is perfectly valid to resort to binary abstractions most of the time, these models come to their limits when synchronization and failures enter the picture: Marino [71] showed that any bistable element, e.g., a binary memory cell, may get stuck arbitrarily long in between its two stable states. This may result in spontaneous, unpredictably late transitions on its output, and even in an inconsistently perceived input at multiple successor gates. While classic designs prevent these scenarios by ensuring that certain timing constraints on the input signals

are not violated, this is not always possible and inherently cannot be assumed in fault-tolerant circuits.

In order to be able to predict the behavior of a circuit and reason formally about its correctness and performance at early design stages, i.e., before fabrication, a suitable circuit model is required. Clearly, any such model should be sufficiently simple to support fast predictions and formal analysis, while at the same time ensure that the results reflect reality sufficiently accurate. We will briefly sketch common approaches.

Discrete time state machines. Synchronously clocked circuits of any kind can be modeled by a discrete-time, discrete-value synchronous communicating state machine model, for which very efficient and fast timing prediction and formal analysis tools are available. Unfortunately, this abstraction does not cover all existing circuits. This is obvious for clockless circuits, but also for the timing analysis of clocked circuits, which is mandatory for validating the clock timing requirements for justifying the synchronous abstraction. The latter is particularly important with the advent of aggressively timed high-speed synchronous circuits, where clock speed must be traded against the increasing rate of manufacturing errors and other sources of timing failures. In that case one has to resort to continuous time models.

Continuous time models. Arguably, the most accurate models used in circuit design today are fully-fledged continuous-time analog valued ones as, e.g., instantiated by Spice [81]. However, excessive analog simulation times prohibit its use for analyzing more than a fairly small part of a VLSI circuit, over fairly short periods of simulated real-time. Discrete-value models, and specifically binary-valued ones, are hence an attractive alternative. Modern digital design approaches e.g. based on description languages such as VHDL [5] incorporate digital timing simulators that are typically based on zero-time Boolean gates interconnected by non-zero-delay channels. Popular instances are simple pure (i.e., constant-delay) and inertial delay channels (i.e., constant-delay channels that suppress short input pulses) [94], but also more elaborate ones like the Delay Degradation Model (DDM) [8] or the empirical Synopsis CCS Timing model [92]. Continuous time, discrete-value models can be either state-based or trace-based, as detailed in the following.

Clockless, state-based models. At the gate level, clockless circuits are typically modeled by a binary state vector representing the global circuit state and, potentially time-annotated, guard-action pairs [4, 72] that describe the gates. An execution, i.e., signal trace, of the circuit is a sequence of global states over time

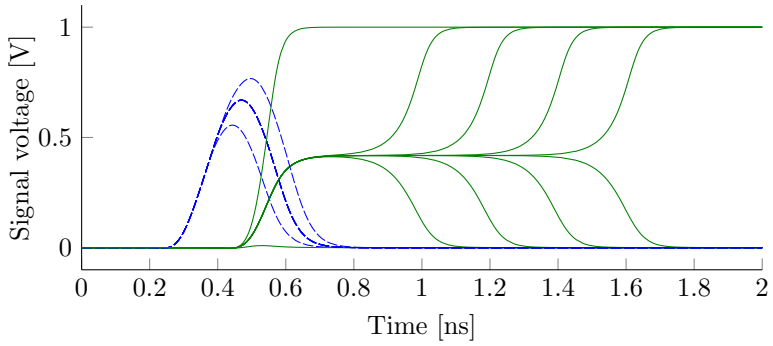

Figure 1: Analog simulation traces of a storage loop (e.g., a 2-input OR gate with the output fed back to one input) that can persistently memorize a high state of its (other) input. The blue dashed curves (the medium line actually corresponding to 8 almost identical pulses) show the real analog shape of short input pulse signals of different duration, the solid green ones give the corresponding output signals of the storage loop.

generated by a parallel execution of the guard-action pairs. Note that executions need not necessarily be unique, accounting for potential delay variations within the circuit. Like the models used in classic distributed computing, such as the Alur-Dill Timed Automata [2] and the Timed I/O Automata by Keynar et al. [55], these models all act on the explicitly given state-space of the underlying system.

While this view serves as a good level of abstraction in clockless circuits operated in closed environments, it comes to its limits when signals do not necessarily stabilize before they change again. For instance, consider a gate that produces a very short low-high-low pulse at its output: In reality, this means that the gate driver circuitry has only started to drive the output signal to high when it is turned off again. This results in a short, potentially non-full-swing waveform that may quite unpredictably affect the subsequent gate. An example is shown in the blue dashed pulse shape in Figure 1.

In this example, the subsequent gate is a *memory flag*, which persistently memorizes a high state at its input, until it is reset again to 0. A straightforward implementation is given by a *storage loop*, e.g. consisting of a 2-input OR gate with its output fed back to its other input. The solid green lines represent the output signals of the storage loop corresponding to the blue dashed inputs. The largest input pulse causes the storage loop to flip to the high state immediately, while the smallest one does not cause any effect on the initial low output. The medium input pulse, however, which actually represents 8 different ones that differ only marginally from each other, causes the loop to enter a metastable state: The input pulses are too short to allow the storage loop, which has some short but non-

zero delay, to settle to a stable value. Depending on minor variations of the input pulses, the metastable state eventually resolves after some unpredictable and possibly large resolution time. The memory flag does not operate as intended during this time, possibly affecting the downstream logic in a similar way.

Such situations cannot be prevented from happening in open environments in which a circuit cannot control all of its inputs. The same holds in fault-tolerant circuits, where the signals provided by faulty nodes may be arbitrary. Thus they must be reasonably covered by an appropriate digital circuit model. Unfortunately, however, this is not the case for any model used in modern timing circuit simulators today. Besides the complete lack of modeling and analysis support for fault-tolerance, it was shown in [48] that none of the existing analytic channel models, including the popular pure and inertial delay channels [94] as well as the DDM model [8], faithfully model the propagation of short pulses in physical circuits. Specifically, it has been shown that these models are inconsistent with possibility and impossibility results concerning the implementation of a one-shot inertial delay channel: a channel that, like an inertial delay channel, suppresses short pulses, but is required to work correctly only in presence of a single input pulse (one-shot).

Recently, however, a candidate delay model [47] based on *involution channels* has been proposed that does not have this shortcoming: It is not only consistent with the theoretical results on the one-shot inertial delay problem [48], but also achieves a promising level of accuracy [82]. As a consequence, there is a prospect of eventually identifying a tractable delay model that can form the basis for a comprehensive modeling framework for digital circuits.

Clockless, trace-based models. Existing frameworks for designing clockless digital circuits also have shortcomings at higher abstraction levels. In particular, we are not aware of any modeling framework (except the one we proposed in [27]) that supports fault-tolerance. Instead of blowing up the state space of existing state-based models – like Alur-Dill Timed Automata [2], Lamport's TLA [63], Timed IO Automatons by Keynar et al. [55], discrete abstractions for hybrid systems [3], or state-space-based control theory [64] – with error states and/or using non-determinism or probabilistic state transitions for modeling faults, it advocates the use of solely trace-based models, which focus on the externally visible behavior of a circuit only.

Examples of earlier trace-based approaches are Ebergen's trace theory for clockless circuits [37] and Broy and Stølen's FOCUS [12]. In both approaches, a module is specified exclusively in terms of the output signal traces that it may exhibit in response to a given input signal trace, without referring to internal state. The trace-based approach introduced in Dolev et al. [27] allows to express tim-

ing conditions via (dense) real-time constraints relating input/output signal transitions, and supports fault-tolerance by allowing (sub-)modules to behave erroneously, i.e., deviate from their specified behavior according to some fault model (e.g. Byzantine behavior [86]). It provides concepts for expressing the composition and implementation of circuits by other circuits, which also allow to rigorously specify self-stabilizing [33] circuits. The model has been used to precisely specify the modules of the Byzantine fault-tolerant and self-stabilizing FATAL clock generation algorithm, which will be described in Section 3.1.4, at a level of abstraction that allows for a direct implementation in hardware.

Compared to state-based approaches, it may be more involved to apply a behavioral approach, in particular, in settings like fully synchronous or fully asynchronous systems, where state-space-based descriptions are reasonably simple. After all, in general, it is even difficult to decide whether behavioral specifications match at interface boundaries [21]. On the other hand, it is much better suited for systems with a complex evolution of the system state over time and/or in which the internal state of system components is too complex or even unknown, which is typically the case for self-stabilizing algorithms. Another attractive side-effect is the inherent support of hierarchical design using (pre-existing or yet to be built) building blocks, without the need to define interface state machines. One can easily build a system and/or its *model* in a modular way, by composing subcomponents and/or their models, whose implementation can be provided (typically by different designers) and verified at a later stage.

Nevertheless, it is understood that behavioral constraints translate into appropriate constraints on the modules' states implicitly. Although making this relation explicit is not part of our modeling framework, this definitely is part of the effort to implement a module and to prove that it indeed exhibits the specified behaviors. The latter may of course involve any appropriate technique, e.g. timed automata [2] and related verification techniques [1].

Open problems. Although the model of [27] explicitly avoids metastable upsets in fault-free executions, it cannot deal explicitly with metastable upsets and metastability propagation. The work by Ginosar [51], which provides several examples of synchronizer circuits where current prediction models drastically underestimate the probability of metastable upsets, shows the importance for such an extension. The challenge here is to bound metastability resolution times and propagation effects, potentially in a probabilistic manner, to be able to quantify upset probabilities and stabilization times.

Besides the need to extend the model [27] by standard tools like simulation relations and abstractions, the integration with a faithful digital circuit model like [82] remains a challenge. The ultimate goal is a comprehensive modeling frame-

work for modern digital circuits, which facilitates (semi-automated) formal verification of circuits, correctness proofs and accurate performance analysis as well as design parameter synthesis, ideally via supporting tools.

3 Generating and Distributing a System Clock

Simulating a synchronous system in the presence of both transient and permanent failures is a challenging task. The traditional approach to generating and distributing the clock signal, a *clock tree* [43], follows the master/slave principle: the signal of a single quartz oscillator is distributed to all logical gates by means of a tree topology. This approach is trivially self-stabilizing, but it must be abandoned due to the possibility of permanent faults; a failure of the tree's root (i.e., the oscillator) or a node close to the root breaks the entire system.

In the severe failure model considered in this article, this fundamental problem was first studied by S. Dolev and Welch [35, 36]. It was motivated by the observation that the assumption of only a fraction of the node being affected by transient faults is too optimistic for the typically long mission times (e.g., space missions) during which clock synchronization has to be provided.

The ultimate goal is to simulate synchronous rounds that are consistently labeled (at all correct nodes) by a round counter modulo $C \geq 2$, where C usually is fairly large. Dolev and Welch give a protocol that stabilizes in exponential time. While this does not seem very exciting at first glance, at this time the big surprise was that the problem was solvable at all!

For the sake of clarity of presentation, let us break down the task into three subproblems:

1. *Pulse Synchronization:* Simulating unlabeled synchronous rounds in a system with bounded communication delay and local clocks of bounded drift.

2. *Counting:* Computing consistent round counters in a synchronous system with unnumbered rounds.

3. *Clock Distribution:* Distributing pulses and/or round numbers efficiently, i.e., using a low-degree topology.

We remark that it is not imperative to follow this structure when solving the problem. However, the discussion will reveal why this is a fairly natural decomposition of the task.

3.1 Pulse Synchronization

In the pulse synchronization problem, we are given a fully connected system of n nodes, $f < n/3$ of which may be Byzantine faulty. Nodes communicate by messages that are delayed between 0 and 1 time units,[2] which also accounts for any delay incurred by local computations. Each node $i \in \{1, \dots, n\}$ is equipped with a local clock $C_i : \mathbb{R}_0^+ \to \mathbb{R}$ of bounded drift, i.e.,

$$\forall t > t' : \; t - t' \leq C_i(t) - C_i(t') \leq \vartheta(t - t')$$

for a constant $\vartheta > 1$. As we require self-stabilization, the initial states of the nodes, including the values of their local clocks, are arbitrary.

Pulse synchronization now requires nodes to regularly trigger *pulses* in a synchronized way. For a time $T \geq 0$, denote by $t_i^{(k)}$ and $i \in \{1, \dots, n\}$, $k \in \mathbb{N}$, the time when node i generates its k^{th} pulse at or after time T (we omit T from the notation). A pulse synchronization algorithm of precision Δ, accuracy bounds $A_{\min}$, $A_{\max}$, and stabilization time S satisfies in any execution that there is some time $T \leq S$ so that

$$\textit{precision: } \forall i, j, k : \; |t_i^{(k)} - t_j^{(k)}| \leq \Delta \text{ and}$$

$$\textit{accuracy: } \forall i, k : \; A_{\min} \leq |t_i^{(k+1)} - t_i^{(k)}| \leq A_{\max} .$$

Here it is implicit that indices i, j refer to correct nodes only, as we cannot make any guarantees on Byzantine nodes' behavior. Note that typically $A_{\min}$ will be a given parameter and the goal is to minimize $A_{\max} - A_{\min}$ as a function of $A_{\min}$ (or vice versa). Due to the drifting local clocks and delayed messages, indistinguishability arguments show that always $\Delta \geq 1$ and $A_{\max} \geq \max\{\vartheta A_{\min}, 1\}$.

3.1.1 Approaches by the Distributed Community

The results from [36] prompted the question whether pulse synchronization could also be solved efficiently, i.e., with a small stabilization time. In a series of papers, the stabilization time was first reduced to polynomial [20] and then linear [18, 29] in n.[3] These works also revealed that randomization is not essential to solve the problem: the latter algorithm is based on running multiple instances of deterministic consensus concurrently.

Together, these results indicated that the problem could admit solutions suitable for hardware implementation. However, none of the above algorithms was a

[2] This is a normalization. In all existing algorithms, the maximum delay affects stabilization times, etc. linearly.

[3] Linear stabilization time was claimed earlier [19], but the algorithm contained non-trivial errors that were fixed in [18].

good candidate, due to unacceptable stabilization time [36], the need for highly accurate local clocks [20], or message size $\Theta(n \log n)$ and too involved local computations [18, 29]. Malekpour provides an alternative linear-time solution [68, 69] with small messages and simple computations, but uses a simplified model (in particular, it is assumed that $\vartheta = 1$, i.e., there is no clock drift).

3.1.2 Approaches by the Hardware Community

Frequently, designers of fault-tolerant architectures consider the clocking mechanism sufficiently reliable and hence do not add any measures for fault-tolerance. The typical rationale is that a clock distribution network has very strong drivers and is therefore not susceptible to transient disturbances. Permanent defects, on the other hand, will make the system stop operation completely, which may be considered safe in some cases. In reality, however, the clock distribution infrastructure is already so complicated that a "partial" defect can easily occur (imagine a driver responsible for a sub-net failing). Moreover, considering that the clock tree is virtually always the most widespread network, showing the highest activity (in terms of transition frequency), it is not so clear why it should not be affected from transient faults as well. These arguments become even more dominant when talking about requirements of, e.g., a failure probability smaller than 10^{-9} per hour. For such a degree of reliability, it is unrealistic to assume that the system can be just "tried out" before being used, and the cost associated with a design error can be extremely high.

As a single clock source like a crystal oscillator constitutes a single point of failure, practitioners aiming for fault-tolerant clocking often turn to the alternative of using multiple clock sources. While this approach is indeed capable of solving the fault-tolerance issue, it at the same time introduces a new problem, namely that of synchronization. In the single-clock case we have a single timing domain to which all activities are synchronized.[4] Within this synchronous domain it is easy to perform communication based on known time bounds, to establish a clear ordering/precedence of events, and to avoid metastability caused by setup/hold time violations (i.e., too short input pulses) at storage elements. When using multiple clock sources, we immediately leave this safe area. It does not matter whether we use multiple clocks with the same nominal frequency or not – the only important distinction is whether the clocks are correlated (i.e., originate at the same source) or uncorrelated. In the latter case, one can never reason about their relative phase (which is essential for avoiding setup/hold time violations), which makes it mandatory to use explicit synchronizers that can, beyond causing performance

[4]The difficulty of providing this time information with the required phase accuracy all over a large system is, besides the fault-tolerance aspect, a key reason why this globally synchronous design paradigm is being challenged.

and area overheads, never attain complete protection from metastable upsets in the general case [60, 71].

With respect to the precision of existing approaches to synchronization, distinguishing "microticks" and "macroticks" has become common. Ultimately, this boils down to dealing with a large value of Δ by dividing the clock frequency (which is between $1/A_{\max}$ and $1/A_{\min}$ in our model) with respect to communication, so that $\Delta \ll 1/A_{\min}$. In other words, slowing down communication sufficiently, one can make the system work despite large Δ. However, this is obviously detrimental to performance, and one hence must strive for minimizing Δ. The software-based clock synchronization mechanisms found in practical applications like the Time-Triggered Protocol TTP [61] or FlexRay [42, 45] rely on adjusting local microtick counters appropriately to attain synchrony on macrotick level. However, both protocols are, essentially, variants of synchronous approximate agreement [32]. Hence, they require pre-established synchrony for correct operation, implying that they are not self-stabilizing.

Modern application-specific integrated circuits (ASICs) are typically composed of many internally synchronous function blocks that "solve" the issue of synchronization in an even simpler way. Instead of relying on any kind of synchronization between different clock domains, these blocks communicate without making any assumptions on timing (one needs still to avoid buffer overflows, however). This paradigm is called *globally asynchronous locally synchronous* (GALS) in the literature [14]. The intention here is mainly to mitigate the clock distribution problem, but this also provides a foundation for establishing fault-tolerance. Due to the consequent existence of multiple clock domains, such architectures feature copious amounts of synchronizers (to avoid metastability) and arbiters (to establish precedence). This works in practice, but comes with the associated performance penalties. Moreover, due to the current lack of tractable models accounting for metastability, there is no satisfying answer to the question "how many synchronizers suffice?"

What is needed to get rid of the performance and area overheads incurred by the GALS approach is correlated clocking all over the system, even if the phase is not perfectly matched in all places. Such clocks are called *mesosynchronous*. Probably the most natural implementation of such a distributed correlated clock source is a ring oscillator. The underlying principle is to use the delay along a cyclic path (gates plus interconnect) as a time reference. More specifically, such a path is implemented in an inverting fashion (odd number of inverting elements), such that the level is periodically inverted with the loop delay defining the half period of the oscillation. Examples of such approaches are the circuits presented by Maza et al, [77] and Fairbanks et al. [38]. They are ring oscillators in that they both exploit the fact that a circle formed by an odd number of inverters will oscillate, and the frequency of the produced clock is determined by circuit delays. In

contrast to the simple basic ring oscillator scheme, multiple "rings" are connected to form meshes of inverters that distribute "clock waves", thereby also generating new ones. In forming the mesh, output signals of inverters are joined by simply hardwiring them and forked by wire forks.

While these approaches can indeed provide a fast clock that is perceived as "correlated" all over the system, the clock is still not intended and claimed to be fault-tolerant by the authors.

3.1.3 DARTS

One may view the above hardware realizations of distributed clock generators as very simple distributed algorithms, in which the algorithmic steps are determined by the laws of superposition at the merging points. From a theoretical point of view, this results in extremely limited options for the designer of the algorithm. Thus, it is quite tempting to try out more sophisticated algorithms and prove strong(er) fault-tolerance properties. To this end, a suitable algorithm must be chosen and the hardwiring be replaced by an implementation based on logic gates.

This idea was at the heart of the DARTS project [50]. The overall goal of the project was to implement the fault-tolerant synchronization algorithm by Srikanth and Toueg [90] in hardware. The pseudo-code of the algorithm, given in Algorithm 1, is executed at each node of a fully connected network of $n > 3f$ nodes, where f is the number of Byzantine faulty nodes it can tolerate. The code is extremely simple, yet one should not be fooled: its implementation in hardware required to overcome non-trivial obstacles [44].

Algorithm 1 Pseudo-code of a node to synchronously generate round(k) messages.

Upon bootup
1: $k \leftarrow 0$;
2: broadcast round(0);
Upon reception of a message
3: **if** received round(ℓ) from at least $f + 1$ distinct nodes with $\ell > k$ **then**
4: broadcast round($k + 1$), ..., round(ℓ);
5: $k \leftarrow \ell$;
6: **end if**
7: **if** received round(k) from at least $2f + 1$ distinct nodes **then**
8: broadcast round($k + 1$);
9: $k \leftarrow k + 1$;
10: **end if**

According to the algorithm's assumptions, a fully connected communication structure was established, which also provides the highest possible degree of fault-tolerance. The implementation of the merging points, i.e., the actual algorithm, was done in clockless logic. This avoids the issue of having to synchronize the local clock source to the correlated "global" time provided by the algorithm (otherwise one would have to rely on synchronizers again), but in turn requires a careful algorithmic design and timing analysis of the entire system [49]. Interestingly, this means that the only timing sources in DARTS are lower and upper bounds on wire delays, without any formal local clocks. Thus, it is quite close in spirit to the solutions inspired by ring oscillators discussed previously. The hardware implementation of a DARTS node is shown in Figure 2.

The implementation of these hardware nodes, which were called "tick generation algorithm (TG-Alg) nodes," was a very insightful example for how difficult it is to map algorithms that were, at best, developed with a software implementation in mind, to hardware. Assumptions that seem simple at the level of an algorithm may turn out extremely painful when having to be realized in hardware. Examples here are the existence of unbounded counters (such as "k" in Algorithm 1), the request for mutual exclusive execution of tasks, the generous use of operators (multiplication is expensive to implement in hardware), etc.

The identification of relevant timing constraints was a challenging issue in the design of the DARTS prototype ASIC as well. Recall that metastability can, in principle, not be avoided (in the general case) for uncorrelated clock sources. However, one can show that in fault-free executions, metastability does not occur. This is not straight-forward due to the following cyclic dependencies: Under the assumption of proper function of the algorithm one can rely on its guaranteed properties (e.g. precision) when establishing the timing closure to avoid setup/hold time violations. In turn, the freedom from setup/hold time violations is a prerequisite for correct functionality.[5] Note that such timing guarantees are essential, as metastability, a possible result of setup/hold violations, results in unspecified behavior not covered by the analysis of the algorithm.

Several key techniques were applied for overcoming the above challenges:

- the use of difference counters for the cycle number, thus mitigating the problem of unlimited counting;

- the implementation of these counters through Muller pipelines, thus avoiding metastability issues that would arise from concurrent increase and decrease operations of the same counter;

[5]For DARTS, "only" an inductive argument was required. When turning to self-stabilization, establishing that the most basic timing assumptions eventually hold tends to be the most difficult aspect of the reasoning.

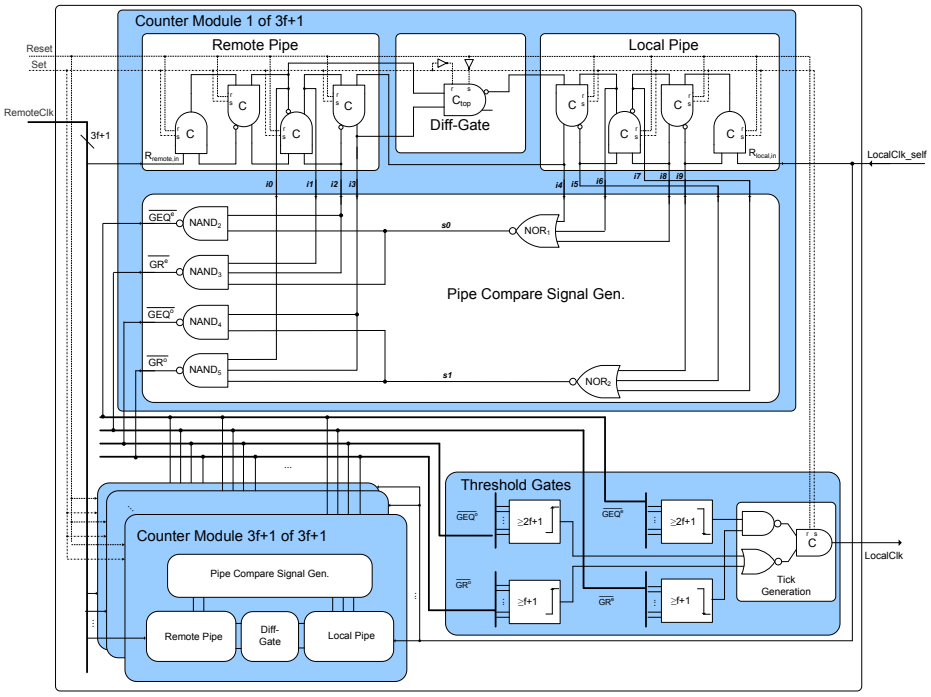

Figure 2: Hardware implementation of a DARTS node in clockless logic.

- a careful mix of event-based and state-based logic;
- the separated treatment of rising and falling signal edges in order to substantially relax the timing constraints.

The project succeeded in developing a working prototype chip, demonstrating that it was indeed feasible to use a distributed algorithm for generating a system-wide clock *and* prove that its *implementation* provides respective guarantees:

- bounded precision and accuracy,
- tolerance of Byzantine faults, and
- use of standard hardware libraries, with one exception: a C-Element must be added.

While the third property might seem like an oddball here, one should be aware that novel circuit components need to be designed on transistor level, layouted, characterized and validated (by simulation) as well. The existing standard libraries had to be augmented by a C-Element for DARTS.

While DARTS constituted a breakthrough in terms of bringing theory and practice closer to each other, the resulting solution exhibits a number of deficiencies calling for further research:

- full connectivity between nodes;
- lack of recovery from transient faults: even if only a minority of nodes undergoes transient faults at any time, there is no mechanism to recover to a correct state;
- too small frequency, limited by the propagation delay of a long loop;
- non-trivial initialization due to fairly strict demands on initial synchrony.

3.1.4 FATAL

In light of the theoretical results from Section 3.1.1 and the proof-of-concept that Byzantine fault-tolerance is viable in low-level hardware provided by DARTS, the obvious next question is whether self-stabilization can be added on the circuit level, too. This was answered affirmatively in [26].

The FATAL algorithm builds on the idea of adding a recovery mechanism to a pulse generation mechanism based on threshold voting. On an abstract level, the FATAL algorithm can be viewed as an implementation of the Srikanth-Toueg algorithm (c.f., Algorithm 1) that avoids having to keep book on tick/pulse numbers by making sure that the time between pulses is sufficiently large: instead of broadcasting round(k) messages, we simply broadcast round messages in Algorithm 1. Another interpretation is that of having a DARTS system that runs slow enough to operate with "pipes of length one", i.e., simple memory cells.

The basic principle is illustrated in Figure 3, depicting a state machine each node runs a copy of. Circles represent states, arrows state transitions that happen when the condition next to the arrow is satisfied, and the box with "propose" on the state transition from *pulse* to *ready* indicates that the nodes' memory flags are reset during this state transition. Each node continuously broadcasts whether it is in state propose or not, and when a node perceives another in this state according to this signal (including itself), its respective memory flag is set (i.e., each node has one memory cell for each node, including itself). The condition "3ϑ local time passed" means that node i will transition from *pulse* to *ready* at the time t when its local clock reads $C_i(t) = C_i(t') + 3\vartheta$, where t' is the time when it switched to *pulse*. Nodes generate a pulse when switching to *pulse*.

It is not hard to verify that, if all nodes start in *ready* with their memory flags cleared, this algorithm will solve pulse synchronization with $\Delta = 2$, $A_{\min} = 3+3\vartheta$, and $A_{\max} = 3+3\vartheta+3\vartheta^2$. By making nodes wait longer in one of the transitions by

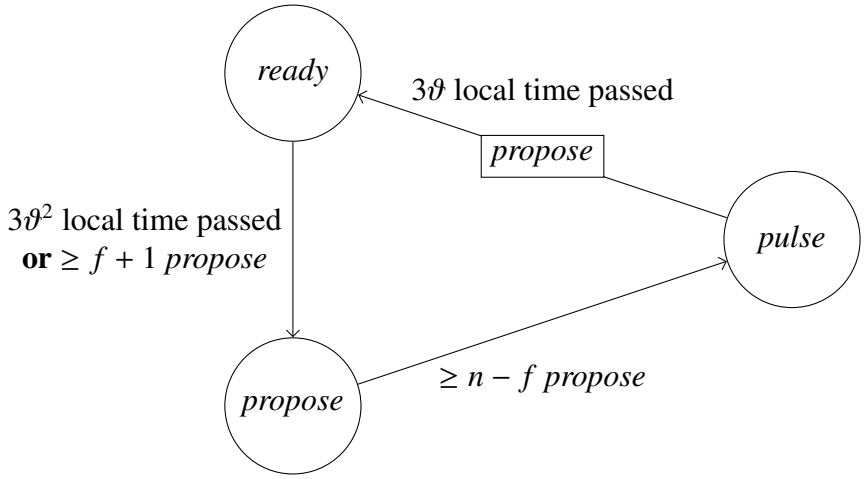

Figure 3: Simple pulse synchronization requiring consistent initialization.

ϑx local time, we can actually have $A_{\min} = 3+3\vartheta+x$ and $A_{\max} = 3+3\vartheta+3\vartheta^2+\vartheta x$, for any $x \geq 0$, i.e., $A_{\max} \to \vartheta A_{\min}$ for $x \to \infty$.

We believe that the recovery mechanism developed for FATAL is a potential key building block of further improvements in the future. In [26], the above basic algorithm is modified so that the task of "stabilizing" the routine, i.e., getting it into a (global) state as if it had been initialized correctly, is reduced to generating a single pulse *by an independent algorithm*. More precisely, all correct nodes need to trigger a "start stabilization" event within a reasonably small time window and then not trigger another such event for $\Theta(1)$ time in order to guarantee stabilization.

The easier task of generating a single "helper pulse" for the purpose of recovery from transient faults is then solved by relying on randomization. The solution used in [25] generates such a pulse with probability $1-2^{-\Omega(n)}$ within $O(n)$ time, resulting in an overall stabilization time of $\Theta(n)$. Hence, the algorithm matches the best known stabilization time of $O(n)$. The improvement lies in the communication complexity and the amount of local computations: Apart from a few memory flags for each other node, each node runs a state machine with a constant number of states; each node continuously broadcasts only a few bits about its own state. Moreover, the algorithm can operate with arbitrary values of ϑ, permitting to use very simple oscillators as local clock sources.

In [25], the approach is implemented and evaluated in hardware. The experiments confirm the theoretical results from [26]. However, the algorithm cannot be used for clocking as-is, for several reasons:

- The algorithm generates pulses every $\Theta(1)$ time, but the involved constants are impractically large. Naive application of the approach would result in

slowing down systems by several orders of magnitude.

- The pulses are anonymous, i.e., the counting problem discussed in the next section needs to be solved.
- The system is fully connected, which is infeasible in large circuits.

These issues will be discussed next.

3.2 Counting

Once pulse synchronization is solved, it can be used to simulate synchronous rounds: One adjusts the accuracy lower bound $A_{\min}$ such that it allows for the maximal sum of the communication delay between neighbors, the local computations for a round, and a safety margin proportional to Δ (recall that a pulse is not issued at all nodes precisely at the same instant of time). However, due to the strong fault model, it is non-trivial to achieve agreement on a round counter. Round counters are highly useful for, e.g., applying pre-determined time division multiple access (TDMA) schemes to shared resources (memory, communication network, etc.) or scheduling synchronized operations (measurements, snapshots, etc.) that are to be executed regularly.

We will now discuss how a self-stabilizing round counter can be constructed in a synchronous system with $f < n/3$ Byzantine faults. The problem of *C-counting*, where C is an integer greater than 2, is formalized as follows. In each round $r \in \mathbb{N}$, each node i outputs a counter $c_i(r) \in \{0, \dots, C-1\}$. The algorithm stabilizes in $S \in \mathbb{N}$ rounds, if for all $r \geq S$ we have

agreement: $\forall i, j : \; c_i(r) = c_j(r)$ and

counting: $\forall i : \; c_i(r+1) = c_i(r) + 1 \bmod C$.

In this subsection, the discussion will be more technical, with the goal of illustrating how the fault-tolerance techniques that have been developed by the distributed computing community are canonical tools for designing fault-tolerant algorithms in hardware. We remark that the inclined reader should feel free to skip the technical proofs, as they are not essential to the remaining discussion in this article.

3.2.1 Equivalence to Consensus

The task of (binary) *consensus* requires that, given an input $b_i \in \{0, 1\}$ at each node at the beginning of round 1, each correct node computes an output o_i satisfying

agreement: $\forall i : o_i = o$ for some $o \in \{0, 1\}$ (we refer to o as the output),

validity: if $\forall i : b_i = b$ then $o = b$, and

termination: all (correct) nodes eventually set their output (exactly once) and terminate.

In practice, one usually requires explicit bounds on when nodes terminate. By an R-round consensus algorithm, we will refer to an algorithm in which all correct nodes terminate by the end of round R.

The counting problem is equally hard as consensus with respect to asymptotic time complexity. We show this for deterministic algorithms and binary consensus algorithms here, but extensions to non-binary consensus and randomized algorithms are straightforward.

Lemma 3.1 (Counting solves consensus)**.** *Given an algorithm for C-counting stabilizing in R rounds, binary consensus (with $f < n/3$ Byzantine nodes) can be solved in R rounds.*

Proof. Denote by $\mathbf{x}(0)$ and $\mathbf{x}(1)$ state vectors such that the counting algorithm would properly count starting from 0 and 1, respectively (regardless of the subset of faulty nodes). Such states must exist, because after stabilization the algorithm will count modulo C and Byzantine nodes may pretend correct operation to avoid detection until after stabilization. Given an input vector $\mathbf{b} \in \{0,1\}^n$, initialize each (correct) node i with state $x_i(b_i)$ and run the algorithm for R rounds. Then each node outputs $c_i(R) - R \bmod 2$.

Clearly, this algorithm terminates in R rounds and, by the agreement property of the counting protocol, all nodes output the same value. Hence it remains to show that this output value is valid, i.e., equals b if $b_i = b$ for all correct nodes. This follows from the choice of $\mathbf{x}(0)$ and $\mathbf{x}(1)$ and the counting property, which implies that, for all correct nodes i, $c_i(R) = R \bmod C$ if $b = 0$ and $c_i(R) = R + 1 \bmod C$ if $b = 1$. □

The other direction was shown in [30]. We present a simpler variant in the following lemma. It makes use of consensus for non-binary values, i.e., $b_i, o_i \in \{0, \dots, C-1\}$ (this case can be reduced to binary consensus in a straightforward manner).

Lemma 3.2 (Consensus solves counting)**.** *Given a synchronous consensus algorithm for inputs $0, \dots, C-1$ terminating in R rounds that tolerates $f < n/3$ Byzantine nodes, C-counting can be solved with stabilization time $O(R)$.*

Proof. Given the consensus algorithm, we solve C-counting as follows. In each synchronous round, we start a new consensus instance that will generate an output value $c_i(r + R)$ at each node i exactly R rounds later (which will double as node

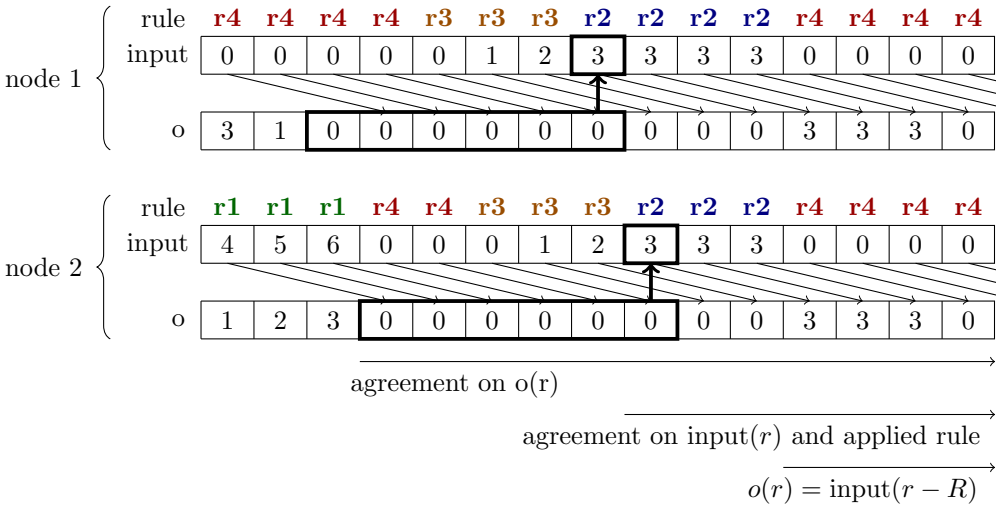

Figure 4: Part of an execution of two nodes running the C-counting algorithm given in the proof of Lemma 3.2, for $C = 8$ and $R = 3$. The execution progresses from left to right, each box representing a round. On top of the input field the applied rule (1 to 4) to compute the input is displayed. Displayed are the initial phases of stabilization: (i) after R rounds agreement on the output is guaranteed by consensus, (ii) then agreement on the input and the applied rule is reached, and (iii) another R rounds later the agreed upon outputs are the agreed upon inputs shifted by 3 rounds.

i's counter value). Note that, while we have no guarantees about the outputs in the first R rounds (initial states are arbitrary), in all rounds $r \geq R$ all correct nodes will output the same value $o(r) = o_i(r)$ (by the agreement property of consensus). Hence, if we define the input value $F_i(r)$ of node i as a function of the most recent $O(R)$ output values at node i, after $O(R)$ rounds all nodes will start using identical inputs $F(r) = F_i(r)$ and, by validity of the consensus algorithm, reproduce these inputs as output R rounds later (cf. Figure 4). In light of these considerations, it is sufficient to determine an input function F from the previous $O(R)$ outputs to values $0, \ldots, C-1$ so that counting starts within $O(R)$ rounds, assuming that the output of the consensus algorithm in round $r + R$ equals the input determined at the end of round r.

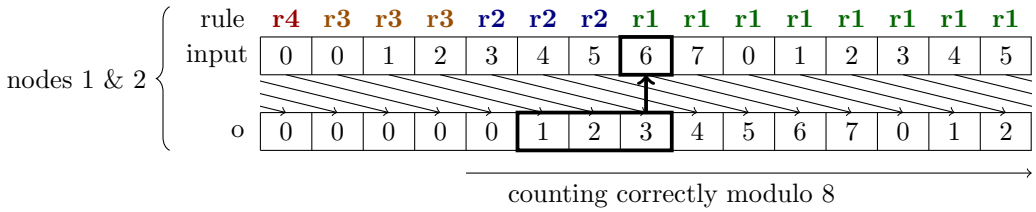

Figure 5: Extension of the execution shown in Figure 4. Nodes have already agreed upon inputs and outputs so that the latter just reproduce the inputs from R rounds ago. The rules now make sure that the nodes start counting modulo 8 in synchrony, always executing rule 1.

We define the following input function, where all values are taken modulo C:

$$\text{input}(r) := \begin{cases} c+R & \text{if } (o(r-R+1),\dots,o(r)) = (c-R+1,\dots,c) \\ x+R & \text{if } (o(r-2R+1-x),\dots,o(r)) = (0,\dots,0,1,\dots,x) \\ & \quad \text{for some } x \in \{0,\dots,R-1\} \\ x & \text{if } (o(r-R+1-x),\dots,o(r)) = (0,\dots,0) \\ & \quad \text{for maximal } x \in \{0,\dots,R-1\} \\ 0 & \text{else .} \end{cases}$$

In the setting discussed above, it is straightforward to verify the following properties of input:

- Always exactly one of the rules applies, i.e., input is well-defined.
- If the outputs counted modulo C for $2R$ consecutive rounds, they will do so forever (by induction, using the first rule); c.f. Figure 5.
- If this does not happen within $O(R)$ rounds, there will be R consecutive rounds where input 0 will be used (by the third and the last rule), c.f. Figure 5.
- Once R consecutive rounds with input 0 occurred, inputs $1,\dots,2R$ will be used in the following $2R$ rounds (by the second and third rule).
- Finally, the algorithm will commence counting correctly (by the first rule).

Overall, if each node i computes its input $F_i(r)$ from its local view of the previous outputs using input, the algorithm will start counting correctly within $S \in O(R)$ rounds. □

These two lemmas imply that there is no asymptotic difference in the round complexity of consensus and the stabilization time of counting. However, note

that the conversion of a consensus algorithm into a counting algorithm given by Lemma 3.2 is very "lossy" in terms of communication complexity and computational efficiency, as R instances of consensus run concurrently! Hence, the main question is whether there are fast solutions to counting that are efficient in terms of communication and computation as well.

3.2.2 Counting Using Shared Coins

The pulse synchronization algorithm by S. Dolev and Welch [36] is conceptually based on a counting algorithm given in the same article, yielding an exponential time randomized solution.

This was improved by Ben-Or et al. [9]. We explain a simpler variant of the algorithm here. The solution is based on *shared coins*. A (weak) shared coin guarantees that there are probabilities $p_0, p_1 > 0$ so that with at least probability p_0, all correct nodes output 0, and with at least probability p_1, all correct nodes output 1. We call $p := \min\{p_0, p_1\}$ the *defiance* of the shared coin. Moreover, we require that the value of the coin is not revealed before the round in which the outputs are generated, so that faulty nodes cannot exploit this knowledge to prevent stabilization.

Observe that we neither require $p_0 + p_1 = 1$ nor that all correct nodes always output the same value. In particular, a trivial shared coin with defiance 2^{-n} is given by each node flipping a coin independently. The algorithm from [36] essentially makes use of this trivial shared coin, which results in its expected exponential stabilization time.

The first step of the algorithm from [9] is to solve 2-counting.

Lemma 3.3 (2-counting from shared coin)**.** *Given a stream of weak shared coins with constant defiance,* 2*-counting can be solved with constant expected stabilization time.*

Proof. In each round r, each node i

1. broadcasts $c_i(r)$;
2. if it received at least $n - f$ times value $c_i(r) - 1 \bmod 2$ in round $r - 1$, it sets $c_i(r + 1) := c_i(r) + 1 \bmod 2$; and
3. otherwise, $c_i(r + 1)$ is set to the output of the shared coin at i in round r.

Before we prove the claim, note that Step 2 depends on messages that were broadcasted in round $r - 1$ instead of messages broadcasted in step 1 during the same round r. The reason for this is to avoid that the faulty nodes exploit so-called *rushing*: As the value of the coin for round r is revealed in round r, faulty

nodes may exploit this information to affect the outcome of the broadcast (in terms of what correct nodes observes) exactly so that in Step 3 the "wrong" action is taken by correct nodes relying on the coin. By referring to the broadcast of the previous round instead, the faulty nodes are forced to commit to an outcome of the broadcast before the coin is revealed, making sure that with probability at least p the "right" action is taken by correct nodes in Step 3.

To see that the algorithm indeed stabilizes, observe first that in cannot happen that, in the same broadcast, a correct node sees value 0 at least $n - f$ times and another correct node sees value 1 at least $n - f$ times: this implies that at least $n - 2f$ correct nodes each have $c_i(r) = 0$ and $c_i(r) = 1$, respectively, but we have only $n - f < n - f + (n - 3f) = 2(n - 2f)$ correct nodes (here we used that $f < n/3$). Assume that $c \in \{0, 1\}$ is the unique value such that some node receives c at least $n - f$ times in round $r - 1$. If there is no such value, choose c arbitrarily. With probability at least p_c, all correct nodes set $c_i(r + 1) := c + 2 \bmod 2 = c$ in round r. Similarly, in round $r + 1$ all nodes set $c_i(r + 2)$ to $c + 1 \bmod 2$ with probability at least p_{1-c}. Once this happened, the clocks of correct nodes will start counting deterministically, as always Step 2 will be executed. Hence, the algorithm stabilizes with (independent) probability $p_0 p_1 \geq p^2$ every other round. □

Once 2-counting is available, the generalization to C-counting is achieved by a similar approach. The key difference is that we now use a two-round protocol controlled by the output of the 2-counting algorithm to solve C-counting. We remark that the algorithm essentially performs a gradecast ([40]) followed by achieving a consistent choice with constant probability using the shared coin.

Lemma 3.4 (C-counting from shared coin and 2-counting)**.** *Given a stream of weak shared coins with constant defiance and a 2-counter, C-counting can be solved with constant expected stabilization time.*

Proof. In each round r, each node i performs the following steps.

1. If the 2-counter reads 0:
   (a) broadcast $c_i(r)$;
   (b) if received value $c \neq \perp$ at least $n - f$ times, set helper variable $h_i(r) := c$, otherwise $h_i(r) := \perp$;
   (c) if $b_i(r-1) = 1$ or the shared coin shows 1 at i in round r, set $c_i(r+1) := c_i(r) + 1 \bmod C$, and otherwise $c_i(r + 1) := 0$.
2. If the 2-counter reads 1:
   (a) broadcast $h_i(r - 1)$;

(b) if received value $c \neq \perp$ at least $n - f$ times, set $c_i(r+1) = c+1 \bmod C$ and $b_i := 1$;

(c) else if received value $c \neq \perp$ at least $n - 2f$ times, set $c_i(r+1) = c+1 \bmod C$ and $b_i(r) := 0$;

(d) else set $c_i(r+1) := 1$ and $b_i(r) := 0$.

To see why this stabilizes with constant probability within $O(1)$ rounds, observe that the following holds once the 2-counter stabilized:

- If in an even round r all correct nodes agree on the clock value and have $b_i(r-1) = 1$, the algorithm will count correctly forever.
- The same holds if they agree on the clock and the shared coin shows 1 at all nodes in round r.
- As $f < n/3$, there can be at most one value $c \neq \perp$ with correct nodes setting $h_i(r) := c$ in an even round r.
- If any correct node i receives this unique value c at least $n - f$ times in the subsequent odd round $r+1$, all correct nodes receive c at least $n - 2f$ times. Hence, *either* it holds that (b) or (c) applies at all correct nodes *or* (c) or (d) apply at all nodes.
- In the first case, all correct nodes have the same clock value. Hence, the shared coin showing 1 in round $r+2$ guarantees stabilization.
- In the second case, all correct nodes set $b_i(r+1) := 0$. Hence, if the coin shows 0 at all nodes in round $r+2$, they all set $c_i(r+3) := 0$ and, subsequently $c_i(r+4) := 1$. If the coin shows 1 at all nodes in round $r+4$, this implies stabilization.

Hence, the algorithm stabilizes with (independent) probability $\min\{p_1, p_0 p_1\} \geq p^2$ within every 4 rounds once the 2-counter counts correctly. □

Composing the two algorithms yields a C-counter with expected constant stabilization time. We stress the similarity of the routine to solutions to consensus based on shared coins [88]; the ideas and concepts developed for consensus translate directly to the counting problem, even if it might be harder in terms of the required communication.

Unfortunately, this approach to solving counting suffers from the same problem as consensus algorithms based on shared coins: theoretically sound protocols that generate shared coins based on the private randomness of the nodes are highly expensive in terms of communication and computation.

3.2.3 Constructing Large Counters from Small Counters

There are several techniques for constructing large counters from small counters, indicating that the key challenge is to obtain a 2-counter. One is given by Lemma 3.4, which however necessitates the presence of a shared coin. Another one is very basic, but inefficient time-wise, as C enters the stabilization time as a factor.

Lemma 3.5 (C-counting from 2-counting [31])**.** *Given a 2-counting algorithm with stabilization time S, for any $k \in \mathbb{N}$ we can solve 2^k-counting with stabilization time $2^k S$ and at most factor 2 more communication.*

Proof. The proof goes by induction over k, the base case being covered by assumption. For the step, we simply execute the 2-counting algorithm slower, by performing one round when the 2^k-counter switches to 0. This way, concatenating the clock bit of the slow 2-counter and the 2^k-counter, we obtain a 2^{k+1}-counter. The stabilization time is $2^k S$ for the slowed-down 2-counter plus the stabilization time of the 2^k-counter, yielding by induction a total stabilization time of $\sum_{l=1}^{k} 2^l S < 2^{k+1} S$. The communication bounds of the 2-counting algorithm together with the slow-down yield the claim concerning the amount of communication.[6] □

A simple variant on the theme achieves faster stabilization at the cost of increased message size.

Lemma 3.6 (C-counting from 2-counting, faster stabilization)**.** *Assuming we are given a 2-counting algorithm with stabilization time S, for any $k \in \mathbb{N}$ we can solve 2^k-counting with stabilization time $2^k + kS$ and at most factor k more communication.*

Proof. The proof goes by induction over k, the base case being covered by assumption. For the step, we execute another copy of the 2-counting algorithm with a minor change: If the already constructed 2^k-counter reads 0, we skip a round of the 2-counting algorithm. Thus, the 2-counter will proceed by $2^k - 1 \bmod 2 = 1$ every 2^k rounds. The 2^{k+1}-counter is now given by the 2^k-counter and an additional leading bit, which is the value the 2-counter had when the 2^k-counter most recently was 0. By the above considerations, the 2^{k+1}-counter will count correctly once (i) both counters stabilized and (ii) the 2^k-counter had value 0 once after this happened.

[6] Note that one can also ensure that the maximum message size does not increase by more than factor 2, by shifting the communication pattern so that no more than 2 instances of the 2-counting algorithm communicate in the same round.

The stabilization time bound now follows: once the 2^k-counter is correctly operating, the 2-counter stabilizes within $S + 1$ rounds, and the 2^k-counter will become 0 again within another 2^k rounds; summation yields $\sum_{l=0}^{k} 2^l + S < 2^{k+1} + (k + 1)S$ rounds for stabilization of the constructed 2^{k+1}-counter. The communication bound is trivially satisfied. □

Even if 2-counting can be solved efficiently, these techniques are slow if C is large. Motivated by this issue, in [46] large clocks are constructed from small ones by encoding clock values over multiple rounds. This enables to increase the clock range exponentially. Specifically, the paper provides two main results. The first is essentially a reduction to consensus (with only one instance running at any given time), and it is similar to the approach taken in Lemma 3.4. The key changes are the following:

1. To enable 1-bit messages, broadcasts of clock values are replaced by $\lceil \log C \rceil$ rounds each in which the clock bits are transmitted sequentially.

2. Instead of relying on shared coins, nodes run an instance of consensus with the variables b_i determined in odd "rounds" as input. The output of the consensus algorithm is used by all nodes to decide whether c (shifted by the number of rounds passed) is the next clock value or 0.

3. In all other rounds, clock values are locally increased by 1 modulo C.

Due to the use of consensus, the correctness argument becomes simpler. If the consensus algorithm outputs 1, there must be a node that used input 1 and therefore received $n - f$ times c in the second broadcast. This implies that all nodes received $n - 2f \geq f + 1$ times c and therefore can determine the new clock value. Otherwise, the algorithm is certain to default to resetting clocks to 0.

This approach replaces the need for a shared coin with the need for an efficient consensus algorithm and a sufficiently large counter. We instantiate the result for the phase king protocol [10] in the following corollary.

Corollary 3.7 (Large counters from small counters and consensus [46])**.** *Given a C-counter for $C \in O(n)$ sufficiently large, a $2^{\Omega(C)}$-counter with stabilization time $O(n)$ can be constructed deterministically using 1-bit broadcast messages.*

Note that one can combine this corollary with Lemma 3.5 or Lemma 3.6 to construct large counters from 2-counters. In [46], a randomized alternative to these lemmas is given that constructs larger counters from an $O(1)$-counter at smaller overhead. Using either of the two lemmas to obtain the required $O(1)$-counter, the following corollary can be derived.

Corollary 3.8 (Large counters from 2-counters using randomization [46])**.** *Given a 2-counter, C-counting can be solved with expected stabilization time $O(n+\log C)$ and $O(\log^* C)$ broadcasted bits per node and round.*

3.2.4 Counting from Pulse Synchronization

Ironically, the obstacle of solving 2-counting disappears if it is feasible to remove one level of abstraction and exert some control over how (perfect) synchrony is simulated. More concretely, in all existing pulse synchronization algorithms one can increase $A_{\min}$ (the minimum time between consecutive pulses) at will, so that $A_{\max}$ grows proportionally. In particular, this can be used to allow for more than a single synchronous round to be simulated in between pulses. Initializing the simple (non-self-stabilizing) pulse synchronization algorithm given in Figure 3 consistently, we thus can allow for sufficient time to generate a tunable number C of "sub-pulses" before the next pulse occurs. Counting locally modulo C by re-initializing the counter to 0 at each pulse and increasing it by 1 on each sub-pulse, we can use the sub-pulses as round delimiters for simulating synchronous rounds with a self-stabilizing C-counter that comes "for free".

To be precise, this counter does not come entirely for free; apart from the additional logic, increasing $A_{\min}$ may also result in increasing the stabilization time of the pulse synchronization algorithm. However, one can obtain a 2-counter or, in fact, any $O(1)$-counter, without asymptotically affecting the stabilization time of the underlying pulse synchronization algorithm. The techniques for constructing larger counters based on small counters given in [46] then can take it from there.

From an abstract perspective, this can be seen as an implementation of the Srikanth-Toueg algorithm [90] that counts only up to $O(1)$ and then is restarted. This approach is followed by FATAL$^+$, an extended version of FATAL analyzed in [26] and implemented and evaluated in [25]. Owing to the simplicity of the algorithm given in Figure 3, the sub-pulses can actually be produced at a higher frequency and with better precision than offered by the basic FATAL algorithm.

We remark that the method of coupling the two algorithms in FATAL$^+$ may be of independent interest. We expect that it can also be applied to couple FATAL or FATAL$^+$ to non-stabilizing pulse synchronization protocols based on approximate agreement, like the earlier discussed TTP and FlexRay protocols. This bears the promise of obtaining a pulse synchronization protocol that (i) can run at even higher frequencies (i.e., $A_{\min}$ is smaller) and (ii) achieves a precision in the order of the *uncertainty* of the communication delay, i.e., if messages are underway between $1-\varepsilon$ and 1 time unit, then $\Delta \in O(\varepsilon)$. This is to be contrasted to algorithms like DARTS or FATAL, which use explicit voting for resynchronization at each pulse and therefore have $\Delta \geq 1$ even if there is a lower bound on the communication delay. Note that the uncertainty of the communication delay is known to

be a lower bound on the worst-case precision of any clock synchronization protocol [67], implying that $\Delta \in \Omega(\varepsilon)$ is unavoidable.

3.2.5 Constructing Counters from Scratch

Modifying the system on such a deep level as how the clock signal is provided may not always be possible, e.g., when one designs a submodule in a larger circuit. In this case, one may still have to solve 2-counting directly.

Recent research has started to tackle this issue. In [31], efficient solutions for the special case of $f = 1$ are designed and proved optimal in terms of the trade-off between stabilization time and number of local states using computer-aided design. However, as the space of algorithms to consider is huge, this method does not scale; even the case $f = 2$ is currently beyond reach.

In [66], a recursive approach is taken to avoid that each node participates in $\Theta(R)$ concurrent instances of an R-round consensus algorithm used for establishing agreement on clock values. The target is to, in each step of the recursion, boost the *resilience* of the protocol to faults, while increasing neither the number of required nodes nor the time for stabilization too much. The result is an algorithm of slightly suboptimal resilience $f < n^{1-o(1)}$, but linear stabilization time $O(f)$ and only $O(\log^2 n / \log\log n + \log C)$ state bits per node. These state bits are broadcasted to all other nodes in each round. For deterministic algorithms, this implies an exponential improvement in communication complexity as compared to the solution from Lemma 3.2, since deterministic consensus algorithms satisfy that $R > f$ (see [41]).

To sketch the idea of the approach, consider a system of n nodes. Each node i runs a 0-resilient C_i-counter (for some C_i we will determine shortly). This is nothing but a local counter modulo C_i: it is increased in each round, and it works correctly so long as i does not fail. We use these counters to let nodes determine temporary leaders that will assist with stabilization if required; once the system is stabilized, the corresponding communication will be ignored. The current leader's local counter is used to provide a temporarily working counter to all nodes. This counter is used to run an $O(f)$-round consensus algorithm, the phase king protocol [10], to agree on the counter values. It is straightforward to show that agreement cannot be destroyed by this mechanism once it is achieved, even if the temporary counter produces garbage later on.

In short, this mechanism reduces the task to ensuring that eventually a correct leader will emerge and persist for $R \in O(f)$ rounds. We achieve this as follows: Node 1 will cycle through all possible leaders, where it keeps "pointing" to the same node for $\Theta(R)$ consecutive rounds. Node 2 does the same, albeit slower by a factor of $2n$. This guarantees that, for any other node j, nodes 1 and 2 eventually consider it the leader for R consecutive rounds. We proceed inductively, slowing

down the "leader pointer" of node j by a factor of $(2n)^{j-1}$ compared to the one of node 1. Clearly, eventually all correct nodes will point to a correct node for R consecutive rounds.

The downside of this approach is that the stabilization time is exponential, since the slowest pointer takes $R \cdot (2n)^n$ rounds to complete a single cycle. Here the recursion comes into play. Instead of using single nodes, on each level of the recursion one groups the nodes into a small number $k \in O(1)$ of clusters. Each cluster runs an f-resilient counter that is used to determine to which leader the node currently points. The "leaders" now are also clusters, meaning that the slowest clock takes $R \cdot (2k)^k \in O(R)$ rounds for a cycle. Now the same principle can be applied, assuming that we can also convince correct nodes in blocks with more than f faults to point to the "correct" leader the blocks with at most f faults will eventually choose. Requiring that fewer than half of the k blocks have more than f faults, this is ensured by an additional majority vote. The resilience of the compound algorithm therefore becomes $\lceil k/2 \rceil (f + 1) - 1$ (one fault less than required to make half of the blocks contain $f + 1$ faults). Crunching numbers and tuning parameters, one obtains the claimed result.

Maybe the most interesting aspect of this construction is its resemblance to recursive approaches for improving the communication complexity of consensus protocols [10, 15, 57]. The additional challenge here is the lack of a common clock, which is overcome by relying on the guarantees from the lower levels together with the simple leader election mechanism explained above. From this point of view, the construction can be interpreted as a natural generalization of the recursive phase king algorithm given in [10]. Accordingly, one may hope that also for counting, it is possible to achieve optimal resilience in a similar way.

3.3 Clock Distribution

All the algorithms so far assume full connectivity, which is unrealistic if the number of nodes is not quite small. In principle, one could seek for solutions to the pulse synchronization and counting problems in low-degree topologies directly. However, it is much easier to solve these tasks in a small, fully connected "core" and then distribute the results redundantly using a sparse interconnection topology. The advantage is that for the distribution problem, it now is sufficient to have pre-defined master/slave relations, i.e., a pre-defined direction of propagation of the clock signal throughout the system. This greatly simplifies the job, as it circumvents the need for reaching any sort of agreement: the clock information is plainly dictated by the nodes further upstream.

When using a sparse network, we must also decrease our expectations in terms of fault-tolerance. Solving clock synchronization (or consensus, for that matter) in the presence of f faults requires minimum node degrees of $2f + 1$, as otherwise

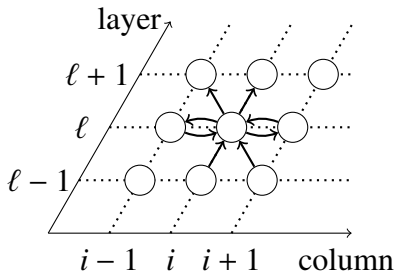

Figure 6: Node i in layer ℓ of a HEX grid and its incident links. The node propagates a pulse when having received it from both nodes on the previous layer. If one fails, the second signal is provided by one of its neighbors in the same layer.

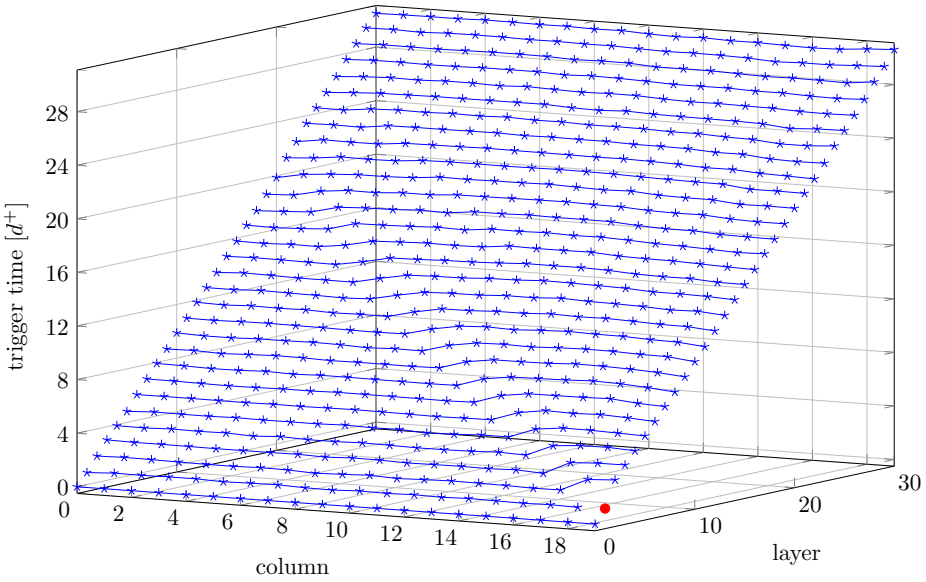

Figure 7: Pulse propagation in HEX with a single Byzantine node. The figure shows pulse trigger times of nodes in a grid: initially nodes (0 to 19) in layer 0 generate pulses in synchrony, feeding these pulses into the grid. The Byzantine faulty node 19 in layer 1 generates a "ripple" in trigger times that is clearly visible in the next layer, but smoothes out over the following layers.

it may happen that a correct node does not have a majority of correct neighbors, rendering it impossible to falsify a claim jointly made by its faulty neighbors [23]. Respecting this, we require only that, for a given parameter f, the system tolerates up to f faulty nodes in the neighborhood of correct nodes.

Following these two paradigms – local fault-tolerance and directed clock propagation – and requiring a "nice" interconnect topology (planarity, connections to physically close nodes) led to HEX [24]. In a HEX grid, each node has 6 neighbors arranged in a hexagon, and the clock information spreads along the layers of the grid, cf. Figure 6. Nodes simply wait for the second signal indicating the current clock pulse, where nodes in the same layer send redundant signals in case one of the nodes in the preceding layer fails. Accordingly, a HEX grid tolerates $f = 1$ local fault, while having small node degrees and a planar topology.[7]

In order to prove precision bounds for HEX, we assumed that communication delays vary by at most $\varepsilon \ll 1$. Clearly, this implies that the precision cannot deteriorate by more than ε per layer. Surprisingly, a much stronger bound of $1 + \Delta_0 + \mathcal{O}(\varepsilon^2 W)$ can be shown on the precision of adjacent nodes, where Δ_0 reflects the precision of the core and W is the width of the HEX grid.

This is an example of a non-trivial *gradient property*, a concept introduced by Fan and Lynch [39]. Finding clock distribution mechanisms that are suitable for hardware realization, fault-tolerant, and provide non-trivial gradient properties is of great interest, as a strong gradient property enables adjacent chip components to communicate at smaller delays in a synchronous, clock-driven fashion. In particular, it directly affects the time required to simulate a synchronous round and hence the operational frequency of the entire system.

Unfortunately, the worst-case precision of HEX deteriorates by $\Theta(f)$ in the presence of f faults, cf. Figure 7. While the simulations show that this is most likely overly pessimistic [65], the adverse effects of even a single fault are problematic in comparison to the surprisingly good performance of the system in absence of faults. We hope that topologies that feature at least $2f + 1$ links to the preceding layer will offer much better precision in face of faults; the idea is illustrated in Figure 8.

Open problems. In Section 3.2.2 we discussed efficient approaches to construct 2-counters from shared coins. While generating shared coins assuming Byzantine failures is prohibitively costly in terms of communication, it is interesting whether there are efficient shared coin protocols under weaker failure assumptions that are realistic for the considered hardware setting.

Clearly, the search for clock distribution topologies that can be implemented

[7] Clearly, the principle can be generalized to larger values of f adding edges, but degrees increase and planarity is lost.

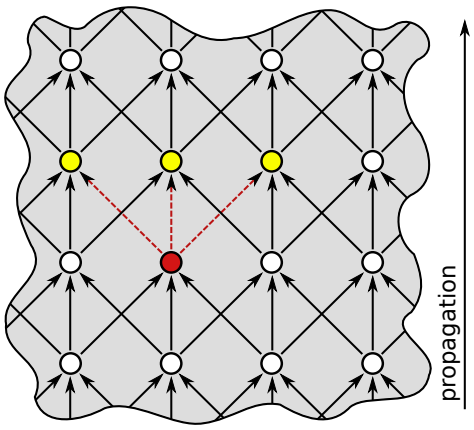

Figure 8: Local structure of a clock propagation approach similar too HEX. Using three connections from the previous layer helps to further mitigate the effect of faults on precision, as the redundant third clock signal does not propagate along a longer path.

with a small number of layers, balanced link delays and sufficiently high degree to tolerate more than 1 local node failure is of central interest. It is also not clear of how to adapt the pulse triggering rules in these HEX-variants to obtain optimal guaranteed precision bounds.

Improving the precision of fault-tolerant self-stabilizing approaches to clocking is an important challenge to achieve utility in practice. As mentioned earlier, it is promising to couple existing solutions with weak precision bounds to algorithms based on approximate agreement to combine high precision and self-stabilization.

Last but not least, an important question is how to verify the correctness of designs prior to production. Striving for algorithms that are sufficiently simple to be implemented in practice bears the promise of enabling formal verification of (parts of) the resulting systems. Given that suitable models can be devised, a grand challenge is the full verification of a fault-tolerant clocking mechanism bottom to top, from gates and wires up to the synchronous abstraction.

4 Conclusion

Due to the continuously increasing scale and complexity of today's VLSI circuits, it becomes insufficient to ensure their reliability by fault mitigation techniques at technological and gate level only, as manufactures will not be able to support the combination of exponentially growing numbers of transistors and decreasing fea-

ture size indefinitely. Error correction, on the other hand, is restricted to storing information, neglecting the issue of dependable computation. Hence, one must strive for algorithmic fault-tolerance above the gate level, but below the abstraction of a synchronous, computationally powerful machine.

The distributed computing community has developed many concepts and algorithmic ideas that can be applied to VLSI circuits once we see them as what they truly are: distributed systems in their own right. The main challenges are

- to adapt and extend the existing theory beyond the abstraction of computationally powerful nodes;
- to devise models of computation that account for faults and metastability in a tractable manner;
- to come up with simple, yet efficient algorithms suitable for hardware implementation; and
- to reason about their correctness in ways ensuring that produced chips *will* work.

We believe that the examples given in this article demonstrate that the existing techniques are essential tools in tackling these challenges. The task that lies ahead is to fill the gap between fault-tolerance in theory and the design of practical, dependable hardware.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical computer science*, 138(1):3–34, Feb. 1995.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, July 2000.

[4] J. H. Anderson and M. G. Gouda. A new explanation of the glitch phenomenon. *Acta Informatica*, 28(4):297–309, 1991.

[5] P. J. Ashenden. *The designer's guide to VHDL*, volume 3. Morgan Kaufmann, 2010.

[6] R. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept. 2005.

[7] S. Beer, R. Ginosar, J. Cox, T. Chaney, and D. M. Zar. Metastability challenges for 65nm and beyond; simulation and measurements. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1297–1302. IEEE, 2013.

[8] M. Bellido, J. Chico, and M. Valencia. *Logic-timing Simulation And the Degradation Delay Model*. Imperial College Press, 2006.

[9] M. Ben-Or, D. Dolev, and E. N. Hoch. Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization. In *27th Symposium on Principles of Distributed Computing (PODC)*, pages 385–394, 2008.

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

[11] G. Brown, M. Gouda, and C. lin Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, Jun 1989.

[12] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag New York, Inc., 2001.

[13] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, 22(4):421–422, 1973.

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

[15] B. A. Coan and J. L. Welch. Modular Construction of a Byzantine Agreement Protocol with Optimal Message Bit Complexity. *Information and Computation*, 97(1):61–85, 1992.

[16] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, 2003.

[17] J. Cortadella and M. Kishinevsky. Synchronous Elastic Circuits with Early Evaluation and Token Counterflow. In *44th Annual Design Automation Conference (DAC)*, pages 416–419, New York, NY, USA, 2007. ACM.

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

[19] A. Daliot, D. Dolev, and H. Parnas. Linear Time Byzantine Self-Stabilizing Clock Synchronization. In *7th International Conference on Principles of Distributed Systems (OPODIS)*, volume 3144 of *LNCS*, pages 7–19. Springer Verlag, Dec 2003. A revised version appears in Cornell ArXiv: *http://arxiv.org/abs/cs.DC/0608096*.

[20] A. Daliot, D. Dolev, and H. Parnas. Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks. In *6th Symposium on Self-Stabilizing Systems (SSS)*, pages 32–48, 2003.

[21] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed Interfaces. In *Embedded Software (EMSOFT)*, pages 108–122, 2002.

[22] A. Dixit and A. Wood. The Impact of New Technology on Soft Error Rates. In *IEEE Reliability Physics Symposium (IRPS)*, pages 5B.4.1–5B.4.7, Apr 2011.

[23] D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms*, 3:14–30, 1982.

[24] D. Dolev, M. Függer, C. Lenzen, M. Perner, and U. Schmid. HEX: Scaling Honeycombs is Easier than Scaling Clock Trees. In *25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[25] D. Dolev, M. Függer, C. Lenzen, M. Posch, U. Schmid, and A. Steininger. Rigorously Modeling Self-Stabilizing Fault-Tolerant Circuits: An Ultra-Robust Clocking Scheme for Systems-on-Chip. *Journal of Computer and System Sciences*, 80(4):860–900, 2014.

[26] D. Dolev, M. Függer, C. Lenzen, and U. Schmid. **F**ault-tolerant **A**lgorithms for **T**ick-generation in **A**synchronous **L**ogic: Robust Pulse Generation. *Journal of the ACM*, 61(5):30:1–30:74, 2014.

[27] D. Dolev, M. Függer, M. Posch, U. Schmid, A. Steininger, and C. Lenzen. Rigorously modeling self-stabilizing fault-tolerant circuits: An ultra-robust clocking scheme for systems-on-chip. *Journal of Computer and System Sciences*, 80(4):860–900, 2014.

[28] D. Dolev, M. Függer, U. Schmid, and C. Lenzen. Fault-tolerant Algorithms for Tick-generation in Asynchronous Logic: Robust Pulse Generation. *Journal of the ACM*, 61(5):30:1–30:74, Sept. 2014.

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

[30] D. Dolev and E. Hoch. On Self-stabilizing Synchronous Actions Despite Byzantine Attacks. In *21st Symposium on Distributed Computing (DISC)*, pages 193–207, 2007.

[31] D. Dolev, J. H. Korhonen, C. Lenzen, J. Rybicki, and J. Suomela. Synchronous Counting and Computational Algorithm Design. In *15th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 237–250, 2013.

[32] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33:499–516, 1986.

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

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

[35] S. Dolev and J. L. Welch. Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults (Abstract). In *14th Symposium on Principles of Distributed Computing (PODC)*, page 256, 1995.

[36] S. Dolev and J. L. Welch. Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. *Journal of the ACM*, 51(5):780–799, 2004.

[37] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[38] S. Fairbanks and S. Moore. Self-Timed Circuitry for Global Clocking. In *11th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 86–96, 2005.

[39] R. Fan and N. Lynch. Gradient Clock Synchronization. In *23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 320–327, 2004.

[40] P. Feldman and S. Micali. Optimal algorithms for Byzantine agreement. In *ACM Symposium on Theory of Computing*, pages 148–161, 1988.

[41] M. J. Fischer and N. A. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters*, 14:183–186, 1982.

[42] FlexRay Consortium et al. FlexRay communications system-protocol specification. *Version 2.1*, 2005.

[43] E. G. Friedman. Clock Distribution Networks in Synchronous Digital Integrated Circuits. *Proceedings of the IEEE*, 89(5):665–692, 2001.

[44] G. Fuchs and A. Steininger. VLSI Implementation of a Distributed Algorithm for Fault-Tolerant Clock Generation. *Journal of Electrical and Computer Engineering*, 2011(936712), 2011.

[45] M. Függer, E. Armengaud, and A. Steininger. Safely Stimulating the Clock Synchronization Algorithm in Time-Triggered Systems – A Combined Formal and Experimental Approach. *IEEE Transactions on Industrial Informatics*, 5(2):132–146, 2009.

[46] M. Függer, M. Hofstätter, C. Lenzen, and U. Schmid. Efficient Construction of Global Time in SoCs despite Arbitrary Faults. In *16th Conference on Digital System Design (DSD)*, pages 142–151, 2013.

[47] M. Függer, R. Najvirt, T. Nowak, and U. Schmid. Towards Binary Circuit Models That Faithfully Capture Physical Solvability. In *Design, Automation, and Test in Europe (DATE)*, 2015.

[48] M. Függer, T. Nowak, and U. Schmid. Unfaithful Glitch Propagation in Existing Binary Circuit Models. In *19th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 191–199, 2013.

[49] M. Függer and U. Schmid. Reconciling Fault-Tolerant Distributed Computing and Systems-on-Chip. *Distributed Computing*, 24(6):323–355, 2012.

[50] M. Függer, U. Schmid, G. Fuchs, and G. Kempf. Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In *6th European Dependable Computing Conference (EDCC)*, pages 87–96, 2006.

[51] R. Ginosar. Fourteen Ways to Fool Your Synchronizer. In *9th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 89–96, 2003.

[52] International Technology Roadmap for Semiconductors, 2012. http://www.itrs.net.

[53] W. Jang and A. J. Martin. SEU-Tolerant QDI Circuits. In *11th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 156–165, 2005.

[54] W. Jang and A. J. Martin. A soft-error-tolerant asynchronous microcontroller. In *13th NASA Symposium on VLSI Design*, 2007.

[55] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.

[56] S. Keller, M. Katelman, and A. J. Martin. A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits. In *15th Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 65–76, 2009.

[57] V. King and J. Saia. Breaking the $O(N^2)$ Bit Barrier: Scalable Byzantine Agreement with an Adaptive Adversary. *Journal of the ACM*, 58(4):18:1–18:24, 2011.

[58] D. J. Kinniment. *Synchronization and Arbitration in Digital Systems*. Wiley, Chichester, 2008.

[59] D. J. Kinniment, A. Bystrov, and A. V. Yakovlev. Synchronization Circuit Performance. *IEEE Journal of Solid-State Circuits*, SC-37(2):202–209, 2002.

[60] L. Kleeman and A. Cantoni. On the Unavoidability of Metastable Behavior in Digital Systems. *IEEE Transactions on Computers*, C-36(1):109–112, 1987.

[61] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[62] I. Koren and Z. Koren. Defect tolerance in VLSI circuits: Techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, Sep 1998.

[63] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[64] E. A. Lee and P. Varaiya. *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, 2nd edition, 2011.

[65] C. Lenzen, M. Perner, M. Sigl, and U. Schmid. Byzantine Self-Stabilizing Clock Distribution with HEX: Implementation, Simulation, Clock Multiplication. In *6th Conference on Dependability (DEPEND)*, 2013.

[66] C. Lenzen, J. Rybicki, and J. Suomela. Towards Optimal Synchronous Counting. In *34th Symposium on Principles of Distributed Computing (PODC)*, 2015.

[67] J. Lundelius and N. Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2-3):190–204, 1984.

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

[69] M. Malekpour. A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol. Technical report, NASA, 2009. TM-2009-215758.

[70] R. Manohar and A. J. Martin. Quasi-delay-insensitive circuits are turing-complete. Technical report, Pasadena, CA, USA, 1995.

[71] L. Marino. General Theory of Metastable Operation. *IEEE Transactions on Computers*, C-30(2):107–115, 1981.

[72] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[73] A. J. Martin. The Limitations to Delay-insensitivity in Asynchronous Circuits. In *Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[74] A. J. Martin. Synthesis of asynchronous VLSI circuits. Technical report, DTIC Document, 2000.

[75] A. J. Martin and M. Nystrom. Asynchronous Techniques for System-on-Chip Design. *Proceedings of the IEEE*, 94(6):1089–1120, Jun 2006.

[76] D. Mavis and P. Eaton. SEU and SET Modeling and Mitigation in Deep Submicron Technologies. In *45th Annual IEEE International Reliability physics symposium*, pages 293–305, April 2007.

[77] M. Maza and M. Aranda. Interconnected Rings and Oscillators as Gigahertz Clock Distribution Nets. In *14th Great Lakes Symposium on VLSI (GLSVLSI)*, pages 41–44, 2003.

[78] M. S. Maza and M. L. Aranda. Analysis of Clock Distribution Networks in the Presence of Crosstalk and Groundbounce. In *International IEEE Conference on Electronics, Circuits, and Systems (ICECS)*, pages 773–776, 2001.

[79] D. G. Messerschmitt. Synchronization in Digital System Design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.

[80] C. Myers and T. H. Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):106–119, June 1993.

[81] L. W. Nagel and D. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, 1973.

[82] R. Najvirt, M. Függer, T. Nowak, U. Schmid, M. Hofbauer, and K. Schweiger. Experimental Validation of a Faithful Binary Circuit Model. 2015. (appears in Proc. GLSVLSI'15).

[83] R. Naseer and J. Draper. DF-DICE: A scalable solution for soft error tolerant circuit design. *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2006.

[84] S. Nassif, K. Bernstein, D. Frank, A. Gattiker, W. Haensch, B. Ji, E. Nowak, D. Pearson, and N. Rohrer. High Performance CMOS Variability in the 65nm Regime and Beyond. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 569–571, Dec 2007.

[85] 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 *19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 174–182, Oct 2004.

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

[87] M. Peercy and P. Banerjee. Fault Tolerant VLSI Systems. *Proceedings of the IEEE*, 81(5):745–758, May 1993.

[88] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

[89] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. *International Conference on Dependable Systems and Networks (DSN)*, pages 389–398, 2002.

[90] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[91] K. Stevens, R. Ginosar, and S. Rotem. Relative timing [asynchronous design]. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, Feb 2003.

[92] Synopsis. CCS Timing. Technical white paper v2.0, 2006.

[93] P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *IEEE Design and Test of Computers*, 24(5):418–428, 2007.

[94] S. H. Unger. Asynchronous Sequential Switching Circuits with Unrestricted Input Changes. *IEEE Transactions on Computers*, 20(12):1437–1444, 1971.

[95] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, 9(3):189–233, 1996.

[96] C. Yeh, G. Wilke, H. Chen, S. Reddy, H. Nguyen, T. Miyoshi, W. Walker, and R. Murgai. Clock Distribution Architectures: a Comparative Study. In *7th Symposium on Quality Electronic Design (ISQED)*, pages 85–91, 2006.

[97] T. Yoneda and C. Myers. Synthesis of Timed Circuits Based on Decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(7):1177–1195, July 2007.

DISTRIBUTED COMPUTING COLUMN
Maurice Herlihy's 60th Birthday Celebration

Panagiota Fatourou
FORTH ICS & University of Crete
faturu@csd.uoc.gr

Maurice Herlihy is one of the most renowned members of the Distributed Computing community. He is currently a professor in the Computer Science Department at Brown University. He has an A.B. in Mathematics from Harvard University, and a Ph.D. in Computer Science from M.I.T. He has served on the faculty of Carnegie Mellon University and on the staff of DEC Cambridge Research Lab. He is the recipient of the 2003 Dijkstra Prize in Distributed Computing, the 2004 Gödel Prize in theoretical computer science, the 2008 ISCA influential paper award, the 2012 Edsger W. Dijkstra Prize, and the 2013 Wallace McDowell award. He received a 2012 Fullbright Distinguished Chair in the Natural Sciences and Engineering Lecturing Fellowship, and he is a fellow of the ACM, a fellow of the National Academy of Inventors, and a member of the National Academy of Engineering and the American Academy of Arts and Sciences.

On the occasion of his 60th birthday, the SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), which was held in Paris, France in July 2014, hosted a celebration which included several technical presentations about Maurice's work by colleagues and friends. This column includes a summary of some of these presentations, written by the speakers themselves. In the first article, Vassos Hadzilacos overviews and highlights the impact of Maurice's seminal paper on wait-free synchronization. Then, Tim Harris provides a perspective on hardware trends and their impact on distributed computing, mentioning several interesting open problems and making connections to Maurice's work. Finally, Michael Scott gives a concise retrospective on transactional memory, another area where Maurice has been a leader. This is a joint column with the Distributed Computing Column of ACM SIGACT News (June 2015 issue), edited by Jennifer Welch. Many thanks to Vassos, Tim, and Michael for their contributions!

A Quarter-Century of Wait-Free Synchronization[1]

Vassos Hadzilacos
Department of Computer Science
University of Toronto, Canada
vassos@cs.toronto.edu

It is an honour and a pleasure to have the opportunity to speak about what in my opinion is Maurice Herlihy's most influential paper, and indeed one of the most significant papers in the theory of distributed computing. I am referring to his work on wait-free synchronization, which appeared in preliminary form in PODC 1988 [8] and in its final form in *TOPLAS* three years later [10]. I will first review the key contributions of this paper and then I will discuss its impact.

1 Review of the key contributions

The context for this work is a distributed system in which processes take steps ***asynchronously*** and communicate by accessing ***shared objects***. Here asynchrony means that between successive steps of a process other processes may take an arbitrary number of steps. Processes are subject to crash failures, meaning that they may stop taking steps altogether, even though they have not reached the end of their computation. For convenience, we assume that a process is designed to take steps (perhaps no-ops) forever, and so we can define a process to have crashed if it takes only finitely many steps in an infinite execution. Minimally, the shared

[1]Remarks on the occasion of Maurice Herlihy's 60th birthday in PODC 2014. Based on the transparencies used in an oral presentation on July 15, 2014, Paris, France. I have tried to preserve the informal tone of that oral presentation here. Supported in part by a grant from the Natural Sciences and Engineering Council of Canada.

objects that the processes use to communicate are registers accessible via separate but atomic write and read operations. The shared objects can also include registers with additional operations such as fetch-and-add, whereby a process atomically increments the value of the register by a specified amount and reads the value of the register before it was incremented; or even other types of shared objects, such as queues or stacks.

The key question that animates the paper is the following:

> "For given object types A and B, in a system with n processes, can we implement an object of type A using objects of type B and registers?"

In what follows, we will take registers (with atomic write and read operations) for granted. So, the above question will be simplified to "in a system of n processes, can we implement an object of type A using objects of type B?"

Here are some specific instances of this question:

- Can we implement a queue shared by two processes using only registers? Herlihy showed that the answer to this question is negative.

- Can we implement a register with a fetch-and-add operation, shared by five processes, using registers with a compare-and-swap operation?[1] Herlihy showed that the answer to this question is affirmative.

What is significant about this paper is not so much the answer to these specific questions, but the tools that it gave us to answer such questions in general.

Having set the general context for the paper, I will now describe its main contributions.

Contribution 1: Model of computation

The type of an object specifies what operations can be applied to an object (of that type) and how the object is supposed to behave ***when operations are applied to it sequentially***. For example, the type queue tells us that we can access the object via enqueue and dequeue operations only, and that in a ***sequence*** of such operations items are dequeued in the order in which they were enqueued. But how should a shared queue behave if operations are applied to it by processes concurrently?

[1] A compare-and-swap operation applied to register X takes two parameters, values *old* and *new*, and has the following effect (atomically): If the present value of X is equal to *old* then X is assigned the value *new* and the value "success" is returned; otherwise, the value of X is not changed and the value "failure" is returned.

More generally, what exactly are the properties that an implementation of an object of type A (shared by n processes) should have? Herlihy requires two properties of such an implementation: linearisability and wait freedom.[2]

Linearisability: The implemented object should behave as if each operation took effect instantaneously, at some point between its invocation and its response.

Wait freedom: An operation on the implemented object invoked by a nonfaulty process eventually terminates, regardless of whether other processes are fast, slow, or even crash.

Note that the requirement of wait freedom implies that implementations

(a) may not use locks: otherwise, a process that crashes while holding a lock could prevent all others from terminating, and

(b) must be starvation free: not only must the system as a whole make progress but each individual nonfaulty process must complete all its operations.

The first important contribution of the paper was the articulation of ***a compelling, elegant, and pragmatic model of computation***.

Contribution 2: Comparing the power of object types

Recall the basic question the paper addresses: In a system of n processes, can we implement an object of type A using objects of type B? An affirmative answer to such a question presents no methodological difficulties: One presents an implementation and a proof that it satisfies the two requisite properties. But what if the answer is negative? How can we prove that A cannot be implemented using B? One way to do so is to show that there is some task C that A can do, and that B cannot do. So, task C is a "yardstick" that can be used to compare A and B. Another key contribution of the paper is ***the identification of the right kind of yardstick*** to compare types, namely, solving the ***consensus problem***. This problem, which under various forms and guises had been studied extensively in fault-tolerant distributed computing before Herlihy's paper, can be described as follows:

- Each process starts with an input.

[2] In my oral presentation, I referred to linearisability as a safety property, and to wait freedom as a liveness property. Rachid Guerraoui, who was in the audience, brought to my attention a paper of his with Eric Ruppert in which they show that this is not quite right [6]: There are types with infinite non-determinism for which linearisability is not a safety property; for types with bounded non-determinism, however, linearisability is indeed a safety property.

- Each nonfaulty process produces an output.
- The output of any process is the input of some process (validity), and is no different than the output of any other process (agreement).

Note that we are interested in ***wait-free*** solutions for this problem. Let us examine some examples of the use of this "yardstick" to prove non-implementability results.

Example 1: To show that in a system of two processes we cannot implement a queue using registers we prove that

(1) using queues we can solve consensus for two processes; and

(2) using registers we cannot solve consensus for two processes.

From (1) and (2) we conclude that we cannot implement queues using only registers: For, if we could, we would combine such an implementation with (1) to obtain an algorithm that uses registers and solves consensus for two processes, contradicting (2).

Example 2: To show that in a system of three processes we cannot implement a register with the compare-and-swap operation using registers with a fetch-and-add operation we prove that

(1) using registers with compare-and-swap we can solve consensus for three processes; and

(2) using registers with fetch-and-add we cannot solve consensus for three processes.

Using similar reasoning as in Example 1, from (1) and (2) we conclude that we cannot implement compare-and-swap using only fetch-and-add.

So, to capture the "power" of an object type A, Herlihy attaches to A a ***consensus number***, namely the unique integer n such that:

- using objects of type A we can solve consensus for n processes, and
- using objects of type A we cannot solve consensus for $n + 1$ processes.

If no such integer n exists, the consensus number of A is ∞. The following, methodologically very useful, theorem follows immediately from this definition.

Theorem 1.1 ([8, 10])**.** *If type A has consensus number n and type B has consensus number $m < n$, then A cannot be implemented from B in a system with more than m processes.*

This leads us to Herlihy's ***consensus hierarchy*** of object types: A type A is at level n of the consensus hierarchy if and only if its consensus number is n — i.e., if and only if A solves consensus for n, but not for $n + 1$, processes. Thus, by Theorem 1.1, "stronger" types are at higher levels of this hierarchy.

Figure 1 illustrates the consensus hierarchy. I now briefly explain the types mentioned in that figure that I have not already defined.

- The test-and-set type (at level 2) refers to a register initialised to 0, with an operation that atomically sets the register to 1 and returns the value of the register before the operation. (So, the operation that is linearised first returns 0, and all others return 1.)

- The n-consensus type (at level n) refers to an object with a PROPOSE(v) operation, where v is an arbitrary value (say a natural number); the object returns the value proposed by the first operation to the first n PROPOSE operations applied to it, and returns an arbitrary value to subsequent operations. (Thus, it is an object designed to solve consensus for n processes.)

- The n-peekable queue type (also at level n) refers to a kind of queue to which a maximum of n values can be enqueued (any values subsequently enqueued are lost) and which allows a process to "peek" at the first value enqueued without dequeuing it.

- The n-assignment type (at level $2n-2$) allows a process to atomically assign n specified values to n specified registers.

- The consensus type (at level ∞) is similar to n consensus, except that it returns to all PROPOSE operations (not only to the first n) the value proposed by the first one.

- Finally, the memory-to-memory swap type (also at level ∞) allows a process to atomically swap the values of two specified registers.

Contribution 3: Universality of consensus

We have seen how Herlihy used consensus as a "yardstick" to compare the relative power of object types. But why is consensus the ***right*** yardstick? In principle, we could have taken any task and used it as a yardstick. For example, consider the ***leader election*** problem:

- Each nonfaulty process outputs "winner" or "loser".

- At most one process outputs "winner".

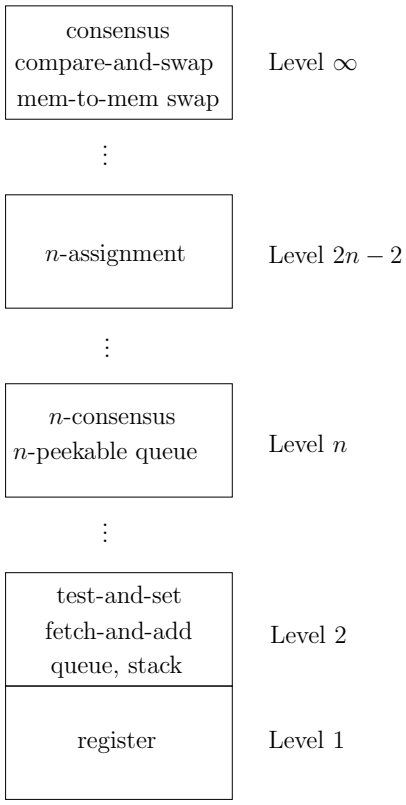

Figure 1: The consensus hierarchy

- Some process outputs "winner" or crashes after taking at least one step.

We could define the "leader election number" of type *A* to be the maximum number of processes for which A can solve the leader election problem — by analogy to the definition of the consensus number, but using a different problem as the yardstick. There is nothing in principle wrong with this, except that the resulting "leader election hierarchy" would not be very interesting: it would consist of just two levels: all types in levels two to infinity of the consensus hierarchy would coalesce into a single level! In other words, unlike consensus, the leader election yardstick is not a very discriminating one. So, what is special about consensus that makes it the right yardstick? The answer lies in the following important fact:

Theorem 1.2 ([8, 10])**.** *Any object type B with consensus number n is* ***universal*** *for n processes: it can implement an object of* ***any*** *type A, shared by n processes.*

The proof of this theorem is through an intricate algorithm that has come to be known as ***Herlihy's universal construction***. Given a function that defines the sequential behaviour of an arbitrary type A, this construction shows how to implement an object of type A shared by n processes using only registers and n-consensus objects. So, given any object of type B with consensus number n, we can solve the consensus problem for n processes (by definition of consensus number), and therefore we can implement n-consensus objects. Then, using Herlihy's universal construction, we can implement an object of type A shared by n processes.

At a very high level, the intuition behind this theorem is simple: Processes use consensus to agree on the order in which to apply their operations on the object they implement. Between this intuition and an actual working algorithm that satisfies wait freedom, however, there is a significant gap. Herlihy's universal construction is an algorithm well worth studying carefully, and returning to every now and then!

2 Impact

The impact of the paper is accurately reflected by its citation count. A Google Scholar search conducted in July 2014 showed over 1400 citations for [10] and over 200 for [8]. Let us look beyond the numbers, however, into the specific ways in which Herlihy's paper on wait-free synchronisation has influenced the field of distributed computing.

Impact 1: The model

The model of asynchronous processes communicating via linearisable, wait-free shared objects that was articulated in a complete form in this paper has been a very influential one. As noted earlier, it is mathematically elegant but also pragmatic. It is certainly true that different aspects of this model appeared earlier, but I believe that this was the first paper that presented the complete package. It is nevertheless useful to trace the heritage.

Shared memory: The asynchronous shared memory model goes back to Dijkstra's seminal paper on mutual exclusion [3].

Wait freedom: The concept of wait-free implementations (though not under this name) originated in Lamport's and Peterson's work on implementations of shared registers [15, 19, 16, 17].

Linearisability: The concept of linearisability as the correctness criterion for the behaviour of shared objects was introduced by Herlihy and Wing [12, 13].

Impact 2: Lock-free data structures

The idea of synchronising access to data structures without relying on locks has had a significant impact on the practice of concurrent programming. Although locking is still (and may well remain) the predominant mechanism employed to coordinate access to data structures by multiple processes, Herlihy's paper helped highlight some of its shortcomings (potential for deadlock, unnecessary restrictions to concurrency, intolerance to even crash failures, priority inversions) and pointed the way to the possibility of synchronising without using locks. There is, by now, an extensive literature on so-called ***lock-free*** data structures. In this context, lock free doesn't necessarily mean wait free. It is a term that encompasses wait freedom as well as the weaker ***non-blocking*** property, which requires that progress be made by ***some*** non-faulty process, not necessarily ***every*** non-faulty process.[3]

Impact 3: Weaker liveness properties

Linearisable wait-free implementations tend to be complex, and one culprit seems to be wait freedom. The most intricate aspect of Herlihy's universal construction is the so-called helping mechanism, which ensures that "no process is left behind". If one is willing to settle for the less demanding non-blocking property, the universal construction becomes much simpler.

The observation that wait freedom seems to complicate things and that it is perhaps too strong a liveness property has led researchers to investigate other liveness properties, weaker than wait freedom, easier to implement, but hopefully still useful in practice. The following are some examples of objects with relaxed liveness requirements:

Obstruction-free objects: Every operation invoked by a nonfaulty process that eventually runs solo (i.e., without interference from other processes) terminates [4, 11].

"Pausable" objects: Every operation invoked by a live process eventually returns control to the caller, either by completing normally, or by aborting without taking effect, or by "pausing" so that another operation can run solo and terminate. An operation can abort or pause only if it encounters interference.

[3]The terms "lock free" and "non-blocking" are not used consistently in the literature; in some papers their meaning is as given here, in others it is reversed.

A nonfaulty process whose operation was paused is required to resume the paused operation and complete it (normally or by aborting) before it can do anything else [2].

Nondeterministic abortable objects: Every operation invoked by a nonfaulty process eventually returns to the caller either by completing normally or by aborting. An operation can abort only if it encounters interference. An aborted operation may or may not have taken effect, and the caller doesn't know which of these two possibilities is the case [1].

Abortable objects: Every operation invoked by a nonfaulty process eventually returns to the caller either by completing normally or by aborting. An operation can abort only if it encounters interference. An aborted operation is guaranteed not to have taken effect [7].

Impact 4: Structure of the "A implemented by B" relation

Though the consensus number of an object type A encapsulates much information about A's ability to implement other types, it does not tell the whole story. By Theorem 1.2, if A has consensus number n, it can support the implementation of any object shared by n processes; but what about the implementation of even "weak" objects, i.e., objects of types whose consensus number is no greater than n, ***shared by more than*** n ***processes***? In this setting, there are phenomena that run counter to the notion that the higher the consensus number of a type the greater its power to implement other types.

Consider the following question: Are all object types at the same level of the consensus hierarchy equivalent? That is, if A and B are two types at the same level n of the consensus hierarchy, can an object of type A, shared by ***any*** number m of processes, be implemented using objects of type B? Or, equivalently (in view of Theorem 1.2), can any object of a type with consensus number n, shared by ***any*** number of processes, be implemented using n-consensus? Herlihy himself proved that this is not the case for level 1: He demonstrated a type at level 1 that cannot be implemented from registers (which are also at level 1) [9]. Rachman proved that this is the case for every level [20]: For every positive integer n, he demonstrated a type T_n at level n of the consensus hierarchy such that an object of type T_n shared by $2n + 1$ processes cannot be implemented using n-consensus objects.[4] (In fact, Rachman's result is more general: for any positive integers n, m such that $m \leq n$, there is a type T_m at level m of the consensus hierarchy such that

[4]My account in this paragraph differs from my oral presentation in Paris, as a result of things I learned in the meanwhile — but should have known then!

an object of type T_m, shared by $2n + 1$ processes, cannot be implemented using n-consensus objects.)

A related set of investigations concern the matter of "robustness" of the consensus hierarchy. Consider a system with n processes. By the definition of consensus number, objects of a type with consensus number less than n cannot implement an n-consensus object. Is it possible, however, to use objects of ***multiple*** "weak" types (with consensus number less than n) to implement n-consensus? If this is possible, we say that the consensus hierarchy is ***not robust***. Jayanti was the first to identify and study the issue of robustness; he proved that under a restricted definition of implementation of one type by others, the consensus hierarchy is not robust [14]. Later, Schenk proved that under a restricted definition of wait freedom, the consensus hierarchy is not robust [21]. Lo and Hadzilacos proved that under the usual definitions of implementation and wait freedom, the consensus hierarchy is not robust [18].

Impact 5: Elevating the status of the bivalency argument

George Pólya and Gabor Szegö made a famous quip about the distinction between a trick and a method:

> "An idea that can be used only once is a trick. If one can use it more than once, it becomes a method." (*Problems and Theorems in Analysis*, 1972.)

Fischer, Lynch, and Paterson gave us the bivalency argument as a brilliant trick in their proof of the impossibility of consensus in asynchronous message-passing systems [5]. With his masterful use of the same argument to prove that consensus among n processes cannot be solved using objects of type B (for several choices of n and B), Herlihy elevated bivalency to the more exalted status of a method!

Impact 6: Design of multiprocessors?

I put a question mark for this impact, because here I am speculating: I do not really know why, in the late 1980s and early 1990s, multiprocessor architects abandoned operations with low consensus number in favour of universal ones. But the timing is such that I wouldn't be surprised to learn that these architects were influenced, at least in part, by Herlihy's discovery that, from the perspective of wait-free synchronisation, much more is possible with operations such as compare-and-swap or load-linked/store-conditional than with operations such as test-and-set or fetch-and-add.

Great papers answer important questions, but also open new ways of thinking, and perhaps even influence practice. Herlihy's paper on wait-free synchronisation delivers on all these counts!

Acknowledgements

I am grateful to Naama Ben-David and David Chan for their comments on this paper.

References

[1] Marcos K. Aguilera, Sven Frolund, Vassos Hadzilacos, Stephanie Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC '07: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 23–32, 2007.

[2] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, pages 122–136, 2005.

[3] Edgar W. Dijkstra. Solution of a problem in concurrent programming control. *Commununications of the ACM*, 8(9):569, 1965.

[4] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, 1998.

[5] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[6] Rachid Guerraoui and Eric Ruppert. Linearizability is not always a safety property. In *Networked Systems - Second International Conference, NETYS 2014*, pages 57–69, 2014.

[7] Vassos Hadzilacos and Sam Toueg. On deterministic abortable objects. In *PODC '13: Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, pages 4–12, 2013.

[8] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, 1988.

[9] Maurice Herlihy. Impossibility results for asynchronous PRAM. In *SPAA '91: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 327–336, 1991.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Maurice Herlihy and Jeannette Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 13–26, New York, NY, USA, 1987. ACM Press.

[13] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[14] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *PODC '93: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–157, 1993.

[15] Leslie Lamport. On concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[16] Leslie Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[17] Leslie Lamport. On interprocess communication. Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[18] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of is: wait-free hierarchies are not robust. In *STOC '97: In Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 579–588, 1997.

[19] Gary Peterson. Concurrent reading while writing. *ACM Transactions of Programming Languages and Systems*, 5(1):46–55, 1983.

[20] Ophir Rachman. Anomalies in the wait-free hierarchy. In *WDAG '94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 156–163, 1994.

[21] Eric Schenk. The consensus hierarchy is not robust. In *PODC '97: In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997.

Hardware Trends: Challenges and Opportunities in Distributed Computing

Tim Harris
Oracle Labs
Cambridge, UK
timothy.l.harris@oracle.com

This article is about three trends in computer hardware, and some of the challenges and opportunities that I think they provide for the distributed computing community. A common theme in all of these trends is that hardware is moving away from assumptions that have often been made about the relative performance of different operations (e.g., computation versus network communication), the reliability of operations (e.g., that memory accesses are reliable, but network communication is not), and even some of the basic properties of the system (e.g., that the contents of main memory are lost on power failure).

Section 1 introduces "rack-scale" systems and the kinds of properties likely in their interconnect networks. Section 2 describes challenges in systems with shared physical memory but without hardware cache coherence. Section 3 discusses non-volatile byte-addressable memory. The article is based in part on my talk at the ACM PODC 2014 event in celebration of Maurice Herlihy's sixtieth birthday.

1 Rack-Scale Systems

Rack-scale computing is an emerging research area concerned with how we design and program the machines used in data centers. Typically, these data centers are built from racks of equipment, with each rack containing dozens of discrete machines. Over the last few years researchers have started to weaken the boundaries

between these individual machines, leading to new "rack-scale" systems. These architectures are being driven by the need to increase density and connectivity between servers, while lowering cost and power consumption.

Different researchers mean somewhat different things by "rack-scale" systems. Some systems are built from existing components. These are packaged together for a particular workload, providing appropriate hardware, and pre-installed software. Other researchers mean systems with internal disaggregation of components: rather than having a rack of machines each with its own network interface and disk, there might be a pool of processor nodes, disk nodes, and networking nodes, all connected over an internal intra-machine interconnect. The interconnect can be configured to connect sets of these resources together in different ways.

Initial commercial systems provide high-density processor nodes connected through an in-machine interconnect to storage devices or to external network interfaces. Two examples are the HP MoonShot [12] and AMD SeaMicro [22] single-box cluster computers. Many further ideas are now being explored in research projects—for instance, the use of custom system-on-chip (SoC) processors in place of commodity chips.

These systems should not just be seen as a way to build a faster data center. Communicating over a modern interconnect is different from communicating over a traditional packet-switched network. Some differences are purely trends in performance—a round-trip latency for over InfiniBand is around 1μs, not much longer than the time it takes to access data stored in DRAM on a large shared-memory multiprocessor. The Scale-Out NUMA architecture provides one example of how latencies may be reduced even further: it exposes the interconnect via a specialized "remote memory controller" (RMC) on a multi-core SoC [18]. Threads in one SoC can instruct the RMC to transfer data to or from memory attached to other processors in the system. Threads communicate with their RMC over memory-mapped queues (held in the SoC's local caches). These operations have much lower latency than accessing a traditional network interface over PCI-express. If network latencies continue to fall, while memory access latencies remain constant, then this will change the optimization goals when designing a protocol.

Other differences are qualitative: as with the Scale-Out NUMA RMC, the main programming interface in many rack-scale systems is RDMA (remote direct memory access). To software, RDMA appears as a transfer from a region of a sender's address space into a region in the receiver's address space. Various forms of control message and notification can be used—e.g., for a receiver to know when data has arrived, or for a sender to know when transmission is complete. Flow control is handled in hardware to prevent packet loss.

Some network devices provide low-latency hardware distribution of data to multiple machines at once (for instance, the ExaLINK matrix switches advertise

5ns latency multicasting data from an input port to any number of output ports [1]). Researchers are exploring how to use this kind of hardware as part of an atomic broadcast mechanism [7].

Research questions: What are the correct communication primitives to let applications benefit from low-latency communication within the system? What are the likely failure modes and how do we achieve fault tolerance? What is the appropriate way to model the guarantees provided by the interconnect fabric in a rack-scale system? How should the interconnect fabric be organized, and how should CPUs, DRAM, and storage be placed in it?

2 Shared Memory Without Cache Coherence

The second trend I will highlight is toward systems with limited support for cache coherence in hardware: Some systems provide shared physical memory, but rely on threads to explicitly flush and invalidate their local caches if they want to communicate through them. Some researchers argue that cache coherence will be provided within a chip, but not between chips [15].

This kind of model is not entirely new. For instance, the Cray T3D system distributed its memory across a set of processor nodes, providing each node with fast access to its local memory, and slower access to uncacheable remote memory [6]. This kind of model makes it important to keep remote memory accesses rare because they will be slow even in the absence of contention (for instance, lock implementations with local spinning are well suited in this setting [16]).

One motivation for revisiting this kind of model is to accommodate specialized processors or accelerators. The accelerator can transfer data to and from memory (and sometimes to and from the caches of the traditional processors) but does not need to participate in a full coherence protocol. A recent commercial example of this kind of system is the Intel Xeon Phi co-processor accessed over PCI-express [13].

A separate motivation for distributing memory is to provide closer coupling between storage and computation. The IRAM project explored an extreme version of this with the processor on the same chip as its associated DRAM [19]. Close coupling between memory and storage can improve the latency and energy efficiency of memory accesses, and permit the aggregate bandwidth to memory to grow by scaling the number of memory-compute modules.

Some research systems eschew the direct use of shared memory and instead focus on programming models based on message passing. Shared memory buffers can be used to provide a high-performance implementation of message passing (for instance, by using a block of memory as a circular buffer to carry messages).

This approach means that only the message passing infrastructure needs to be aware of the details of the memory system. Also, it means that software written for a genuinely distributed environment is able to run correctly (and hopefully more quickly) in an environment where messages stay within a machine.

Systems such as K2 [14] and Popcorn [4] provide abstractions to run existing shared-memory code in systems without hardware cache coherence, using ideas from distributed shared memory systems.

Conversely, the Barrelfish [5] and FOS [23] projects have been examining the use of distributed computing techniques within an OS. Barrelfish is an example of a *multikernel* in which each core runs a separate OS kernel, even when the cores operate in a single cache-coherent machine. All interactions between these kernels occur via message-passing. This design avoids the need for shared-memory data structures to be managed between cores, enabling a single system to operate across coherence boundaries. While it is elegant to rely solely on message passing, this approach seems better suited to some workloads than to others—particularly when multiple hardware threads share a cache, and could benefit from spatial and temporal locality in the data they are accessing.

Research questions: What programming models and algorithms are appropriate for systems which combine message passing with shared memory? To what extent should systems with shared physical memory (without cache coherence) be treated differently from systems without any shared memory at all?

3 Non-Volatile Byte-Addressable Memory

There are many emerging technologies that provide non-volatile byte-addressable memory (NV-RAM). Unlike ordinary DRAM, memory contents are preserved on power loss. Unlike traditional disks, locations can be read or written at a fine granularity—nominally individual bytes, although in practice hardware will transfer complete cache lines. Furthermore, unlike a disk, these reads and writes may be performed by ordinary memory access instructions (rather than using RDMA, or needing the OS to orchestrate block-sized transfers to or from a storage device).

This kind of hardware provides the possibility of an application keeping all of its data structures accessible in main memory. Researchers are starting to explore how to model NV-RAM [20]. Techniques from non-blocking data structures provide one starting point for building on NV-RAM. A power loss can be viewed as a failure of all of the threads accessing a persistent object. However, there are several challenges which complicate matters:

First, the memory state seen by the threads before the power loss is not necessarily the same as the state seen after recovery. This is because, although the

NV-RAM is persistent, the remainder of the memory system may hold data in ordinary volatile buffers such as processor caches and memory controllers. When power is lost, some data will transiently be in these volatile buffers. Aggressively flushing every update to NV-RAM may harm performance. Some researchers have explored flushing updates upon power-loss, but that approach requires careful analysis to ensure that there is enough residual power to do so [17].

The second problem is that applications often need to access several structures—for instance, removing an item from one persistent collection object, processing it, and adding it to another persistent collection. If there is a power loss during the processing step, then we do not want to lose the item.

Transactions provide one approach for addressing these two problems. It may be possible to optimize the use of cache flush/invalidate operations to ensure that data is genuinely persistent before a transaction commits, while avoiding many individual flushes while the transaction executes. As with transactional memory systems, transactions against NV-RAM would provide a mechanism for composing operations across multiple data structures [10]. What is less clear is whether transactions are appropriate for long-running series of operations (such as the example of processing an object when moving it between persistent collections).

Having an application's data structures in NV-RAM could be a double-edged sword. It avoids the need to define translations between on-disk and in-memory formats, and it avoids the time taken to load data into DRAM for processing. This time saving is significant in "big data" applications, not least when restarting a machine after a crash. However, explicit loading and saving has benefits as well as costs: It allows in-memory formats to change without changing the external representation of data. It allows external data to be processed by tools in a generic way without understanding its internal formats (backup, copying, de-duplication, etc.). It provides some robustness against transient corruption of in-memory formats by restarting an application and re-loading data.

It is difficult to quantify how significant these concerns will be. Earlier experience with persistent programming languages explored many of these issues [3]. Recent work on dynamic software updates is also relevant (e.g., Arnold and Kaashoek in an OS kernel [2], and Pina *et al.* in applications written in Java [21]).

Research questions: How should software manage data held in NV-RAM, and what kinds of correctness properties are appropriate for a data structure that is persistent across power loss?

4 Discussion

This article has touched on three areas where developments in computer hardware are changing some of the traditional assumptions about the performance and behavior of the systems we build on.

Processor clock rates are not getting significantly faster (and, many argue, core counts are unlikely to increase much further [9]). Nevertheless, there are other ways in which system performance can improve such as by integrating specialized cores in place of general-purpose ones, or by providing more direct access to the interconnect, or by removing the need to go through traditional storage abstractions to access persistent memory.

I think many of these trends reflect a continued blurring of the boundaries between what constitutes a "single machine" versus what constitutes a "distributed system". Reliable interconnects are providing hardware guarantees for message delivery, and in some cases this extends to guarantees about message ordering as well even in the presence of broadcast and multicast messages. Conversely, the move away from hardware cache coherence within systems means that distributed algorithms become used in systems which look like single machines—e.g., in the Hare filesystem for non-cache-coherent multicores [8].

Many of these hardware developments have been proceeding ahead of the advancement of formal models of the abstractions being built. Although the use of verification is widespread at low levels of the system – especially in hardware – I think there are important opportunities to develop new models of the abstractions exposed to programmers. There are also opportunities to influence the direction of future hardware evolution—perhaps as with how the identification of the consensus hierarchy pointed to the use of atomic compare and swap in today's multiprocessor systems [11].

References

[1] EXALINK Fusion (web page). Apr. 2015. `https://exablaze.com/exalink-fusion`.

[2] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. 4th European Conference on Computer Systems (EuroSys)*, pages 187–198, 2009.

[3] M. Atkinson and M. Jordan. A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, University of Glasgow, Department of Computing Science, 2000.

[4] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *EuroSys '15: Proc. 10th European Conference on Computer Systems (EuroSys)*, page 29, 2015.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP '09: Proc. 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[6] Cray Research Inc. *CRAY T3D System Architecture Overview Manual*. 1993. `ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html`.

[7] M. P. Grosvenor, M. Fayed, and A. W. Moore. Exo: atomic broadcast for the rack-scale computer. 2015. `http://www.cl.cam.ac.uk/~mpg39/pubs/workshops/wrsc15-exo-abstract.pdf`.

[8] C. Gruenwald III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *EuroSys '15: Proc. 10th European Conference on Computer Systems*, page 30, 2015.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[10] T. Harris, M. Herlihy, S. Marlow, and S. Peyton Jones. Composable memory transactions. In *PPoPP '05: Proc. 10th Symposium on Principles and Practice of Parallel Programming*, June 2005.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[12] HP Moonshot system: a new class of server. `http://www.hp.com/go/moonshot`, Accessed 9 July 2014.

[13] Intel Corporation. Intel Xeon Phi coprocessor system software developers guide. 2012. IBL Doc ID 488596.

[14] F. X. Lin, Z. Wang, and L. Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *ASPLOS '14: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–300, 2014.

[15] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[16] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[17] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS '12: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[18] S. Novaković, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-Out NUMA. In *ASPLOS '14: Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[19] D. A. Patterson, K. Asanovic, A. B. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. E. Kozyrakis, D. B. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. A. Yelick. Intelligent RAM (IRAM): the industrial setting, applications and architectures. In *Proceedings 1997 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '97, Austin, Texas, USA, October 12-15, 1997*, pages 2–7, 1997.

[20] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[21] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA '14: Proc. Conference on Object-Oriented Programming Languages, Systems, and Applications*, Oct. 2014.

[22] A. Rao. SeaMicro SM10000 system overview, June 2010. `http://www.seamicro.com/sites/default/files/SM10000SystemOverview.pdf`.

[23] D. Wentzlaff and A. Agarwal. Factored operating systems (FOS): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.

Transactional Memory Today[1]

Michael Scott
Computer Science Department
University of Rochester, NY, USA
scott@cs.rochester.edu

It was an honor and a privilege to be asked to participate in the celebration, at PODC 2014, of Maurice Herlihy's many contributions to the field of distributed computing—and specifically, to address the topic of transactional memory, which has been a key component of my own research for the past decade or so.

When introducing transactional memory ("TM") to people outside the field, I describe it as a sort of magical merger of two essential ideas, at different levels of abstraction. First, at the language level, TM allows the programmer to specify that certain blocks of code should be atomic without saying how to *make* them atomic. Second, at the implementation level, TM uses speculation (much of the time, at least) to execute atomic blocks in parallel whenever possible. Each dynamic execution of an atomic block is known as a *transaction*. The implementation guesses that concurrent transactions will be mutually independent. It then monitors their execution, backing out and retrying if (and hopefully only if) they are discovered to conflict with one another.

The second of these ideas—the speculative implementation—was the focus of the original TM paper, co-authored by Maurice with Eliot Moss [22]. The first idea—the simplified model of language-level atomicity—is also due largely to Maurice, but was a somewhat later development.

[1]Based on remarks delivered at the Maurice Herlihy 60th Birthday Celebration, Paris, France, July 2014

1 Motivation

To understand the original motivation for transactional memory, consider the typical method of a nonblocking concurrent data structure. The code is likely to begin with a "planning phase" that peruses the current state of the structure, figuring out the operation it wants to perform, and initializing data—some thread-private, some visible to other threads—to describe that operation. At some point, a critical *linearizing instruction* transitions the operation from "desired" to "performed." In some cases, the identity of the linearizing instruction is obvious in the source code; in others it can be determined only by reasoning in hindsight over the history of the structure. Finally, the method performs whatever "cleanup" is required to maintain long-term structural invariants. Nonblocking progress is guaranteed because the planning phase has no effect on the logical state of the structure, the linearizing instruction is atomic, and the cleanup phase can be performed by any thread—not just the one that called the original operation.

Two issues make methods of this sort very difficult to devise. The first is the need to effect the transition from "desired" to "performed" with a single atomic instruction. The second is the need to plan correctly in the face of concurrent changes by other threads. By contrast, an algorithm that uses a coarse-grained lock faces neither of these issues: writes by other threads will never occur in the middle of its reads; reads by other threads will never occur in the middle of its writes.

2 The Original Paper

While Maurice is largely celebrated for his theoretical contributions, the original TM paper was published at ISCA, the leading architecture conference, and was very much a hardware proposal. We can see this in the subtitle—"Architectural Support for Lock-Free Data Structures"—and the abstract: "[TM is] ... intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion."

The core idea is simple: a transaction runs almost the same code as a coarse-grain critical section, but with special load and store instructions, and without the actual lock. The special instructions allow the hardware to track conflicts between concurrent transactions. A special end-of-transaction commit instruction will succeed (and make transactionally written values visible to other threads) only if no concurrent conflicting transaction has committed. Here "conflict" means that one transaction writes a cache line that another reads or writes. Within a transaction, a special validate instruction allows code to determine whether it still has a chance to commit successfully—and in particular, whether the loads it has

performed to date remain mutually consistent. In response to a failed validate or commit, the typical transaction will loop back (in software) and start over.

Looking back with the perspective of more than 20 years, the original TM paper appears remarkably prescient. Elision of coarse-grain locks remains the principal use case for TM today, though the resulting algorithms are "lock-free" only in the informal sense of "no application-level locks," not in the sense of livelock-free. Like almost all contemporary TM hardware, Herlihy & Moss (H&M) TM was also a "best-effort-only" proposal: a transaction could fail due not only to conflict or to overflow of hardware buffers, but to a variety of other conditions—notably external interrupts or the end of a scheduling quantum. Software must be prepared to fall back to a coarse-grain lock (or some other hybrid method) in the event of repeated failures.

Speculative state (the record of special loads and stores) in the H&M proposal was kept in a special "transactional cache" alongside the "regular" cache (in 1993, processors generally did not have multiple cache layers). This scheme is still considered viable today, though commercial offerings vary: the Intel Haswell processor leverages the regular L1 data cache [40]; Sun's unreleased Rock machine used the processor store buffer [10]; IBM's zEC12 uses per-core private L2s [25].

In contrast with current commercial implementations, H&M proposed a "responder wins" coherence strategy: if transaction *A* requested a cache line that had already been speculatively read or written by concurrent transaction *B*, *B* would "win" and *A* would be forced to abort. Current machines generally do the opposite: "responder loses"—kill *B* and let *A* continue. Responder-loses has the advantage of compatibility with existing coherence protocols, but responder-wins turns out to be considerably less vulnerable to livelock. Nested transactions were not considered by H&M, but current commercial offerings address them only by counting, and subsuming the inner transactions in the outer: there is no way to abort and retry an inner transaction while keeping the outer one live.

Perhaps the most obvious difference between H&M and current TM is that the latter uses "modal" execution, rather than special loads and stores: in the wake of a special tm-start instruction, all ordinary memory accesses are considered speculative. In keeping with the technology of the day, H&M also assumed sequential consistency; modern machines must generally arrange for tm-start and commit instructions to incorporate memory barriers.

While designers of modern systems—both hardware and software—think of speculation as a fundamental design principle—comparable to caching in its degree of generality—this principle was nowhere near as widely recognized in 1993. In hindsight, the H&M paper (which doesn't even mention the term) can be seen not only as the seminal work on TM, but also as a seminal work in the history of speculation.

3 Subsequent Development

Within the architecture community, H&M TM was generally considered too ambitious for the hardware of the day, and was largely ignored for a decade. There was substantial uptake in the theory community, however, where TM-like semantics were incorporated into the notion of universal constructions [3, 5, 24, 28, 35]. In 1997, Shavit and Touitou coined the term "Software Transactional Memory," in a paper that shared with H&M the 2012 Dijkstra Prize [33].

And then came multicore. With the end of uniprocessor performance scaling, the difficulty of multithreaded programming became a sudden and pressing concern for researchers throughout academia and industry. And with advances in processor technology and transistor budgets, TM no longer looked so difficult to implement. Near-simultaneous breakthroughs in both software and hardware TM were announced by several groups in the early years of the 21st century.

Now, another decade on, perhaps a thousand TM papers have been published (including roughly a third of my own professional output). Plans are underway for the 10th annual ACM TRANSACT workshop. Hardware TM has been incorporated into multiple "real world" processors, including the Azul Vega 2 and 3 [7]; Sun Rock [10]; IBM Blue Gene/Q [36], zEnterprise EC12 [25], and POWER8 [6]; and Intel Haswell [40]. Work on software TM has proven even more fruitful, at least from a publications perspective: there are many more viable implementation alternatives—and many more semantic subtleties—than anyone would have anticipated back in 2003. TM language extensions have become the synchronization mechanism of choice in the Haskell community [16], official extensions for C++ are currently in the works (a preliminary version [1] already ships in `gcc`), and research-quality extensions have been developed for a wide range of other languages.

4 Maurice's Contributions

Throughout the history of TM, Maurice has remained a major contributor. The paragraphs here touch on only a few of his many contributions. With colleagues at Sun, Maurice co-designed the DSTM system [18], one of the first software TMs with semantics rich enough—and overheads low enough—to be potentially acceptable in practice. Among its several contributions, DSTM introduced the notion of out-of-band *contention management*, a subject on which Maurice also collaborated with colleagues at EPFL [13, 14]. By separating safety and liveness, contention managers simplify both STM implementation and correctness proofs.

In 2005, Maurice collaborated with colleagues at Intel on mechanisms to virtualize hardware transactions, allowing them to survive both buffer overflows and

context switches [30]. He also began a series of papers, with colleagues at Brown and Swarthmore, on transactions for energy efficiency [12]. With student Eric Koskinen, he introduced *transactional boosting* [20], which refines the notion of conflict to encompass the possibility that concurrent operations on abstract data types, performed within a transaction, may commute with one another at an abstract level—and thus be considered non-conflicting—even when they would appear to conflict at the level of loads and stores. With student Yossi Lev he explored support for debugging of transactional programs [21]. More recently, again with the team at Sun, he has explored the use of TM for memory management [11].

Perhaps most important, Maurice became a champion of the promise of transactions to simplify parallel programming—a promise he dubbed the "transactional manifesto" [19]. During a sabbatical at Microsoft Research in Cambridge, England, he collaborated with the Haskell team on their landmark exploration of *composability* [16]. Unlike locks, which require global reasoning to avoid or recover from deadlock, transactions can easily be combined to create larger atomic operations from smaller atomic pieces. While the benefits can certainly be oversold (and have been—though not by Maurice), composability represents a fundamental breakthrough in the creation of concurrent abstractions. Prudently employed, transactions can offer (most of) the performance of fine-grain locks with (most of) the convenience of coarse-grain locks.

5 Status and Challenges

Today hardware TM appears to have become a permanent addition to processor instruction sets. Run-time systems that use this hardware typically fall back to a global lock in the face of repeated conflict or overflow aborts. For the overflow case, hybrid systems that fall back to software TM may ultimately prove to be more appropriate. STM will also be required for TM programs on legacy hardware. The fastest STM implementations currently slow down critical sections (though not whole applications!) by factors of 3–5, and that number is unlikely to improve. With this present status as background, the future holds a host of open questions.

5.1 Usage Patterns

TM is not yet widely used. Most extant applications are actually written in Haskell, where the semantics are unusually rich but the implementation unusually slow. The most popular languages for research have been C and C++, but progress has been impeded, at least in part, by the lack of high quality benchmarks.

The biggest unknown remains the breadth of TM applicability. Transactions are clearly useful—from both a semantic and a performance perspective—for small operations on concurrent data structures. They are much less likely to be useful—at least from a performance perspective—for very large operations, which may overflow buffer limits in HTM, run slowly in STM, and experience high conflict rates in either case. No one is likely to write a web server that devotes a single large transaction to each incoming page request. Only experience will tell how large transactions can become and still run mostly in parallel.

When transactions *are* too big, and frequently conflict, programmers will need tools to help them identify the offending instructions and restructure their code for better performance. They will also need advances, in both theory and software engineering, to integrate transactions successfully into pre-existing lock-based applications.

5.2 Theory and Semantics

Beyond just atomicity, transactions need some form of condition synchronization, for operations that must wait for preconditions [16, 37]. There also appear to be cases in which a transaction needs some sort of "escape action" [29], to generate effects (or perhaps to observe outside state) in a way that is not fully isolated from action in other threads. In some cases, the application-level logic of a transaction may decide it needs to abort. If the transaction does not restart, but switches to some other code path, then information (the fact of the abort, at least) has "leaked" from code that "did not happen" [16]. Orthogonally, if large transactions prove useful in some applications, it may be desirable to parallelize them internally, and let the sub-threads share speculative state [4]. All these possibilities will require formalization.

A more fundamental question concerns the basic model of synchronization. While it is possible to define the behavior of transactions in terms of locks [27], with an explicit notion of abort and rollback, such an approach seems contrary to the claim that transactions are simpler than locks. An alternative is to make atomicity itself the fundamental concept [8], at which point the question arises: are aborts a part of the language-level semantics? It's appealing to leave them out, at least in the absence of a program-level abort operation, but it's not clear how such an approach would interact with operational semantics or with the definition of a data race.

For run-time–level semantics, it has been conventional to require that every transaction—even one that aborts—see a single, consistent memory state [15]. This requirement, unfortunately, is incompatible with implementations that "sandbox" transactions instead of continually checking for consistency, allowing doomed transactions to execute—at least for a little while—down logically impossible

code paths. More flexible semantics might permit such "transactional zombies" while still ensuring forward progress [32].

5.3 Language and System Integration

For anyone building a TM language or system, the theory and semantic issues of the previous section are of course of central importance, but there are other issues as well. What should be the syntax of atomic blocks? Should there be atomic expressions? How should they interact with existing mechanisms like try blocks and exceptions? With locks?

What operations can be performed inside a transaction? Which of the standard library routines are on the list? If routines must be labeled as "transaction safe," does this become a "viral" annotation that propagates throughout a code base? How much of a large application must eschew transaction-unsafe operations?

In a similar vein, given the need to instrument loads and stores inside (but not outside) transactions, which subroutines must be "cloned"? How does the choice interact with separate compilation? How do we cope with the resulting "code bloat"?

Finally, what should be done about repeated aborts? Is fallback to a global lock acceptable, or do we need a hybrid HTM/STM system? Does the implementation need to adapt to observed abort patterns, avoiding fruitless speculation? What factors should influence adaptation? Should it be static or dynamic? Does it need to incorporate feedback from prior executions? How does it interact with scheduling?

5.4 Building and Using TM Hardware

With the spread of TM hardware, it will be increasingly important to use that hardware well. In addition to tuning and adapting, we may wish to restructure transactions that frequently overflow buffers. We might, for example—by hand or automatically—reduce a transaction's memory footprint by converting a read-only preamble into explicit (nontransactional) speculation [2, 39]. One of my students has recently suggested using advisory locks (acquired using nontransactional loads and stores) to serialize only the portions of transactions that actually conflict [38].

Much will depend on the evolution of hardware TM capabilities. Nontransactional (but immediate) loads and stores are currently available only on IBM Power machines, and there at heavy cost. Lightweight implementations would enable not only partial serialization but also ordered transactions (i.e., speculative parallelization of ordered iteration) and more effective hardware/software

hybrids [9, 26]. As noted above, there have been suggestions for "responder-wins" coherence, virtualization, nesting, and condition synchronization. With richer semantics, it may also be desirable to "deconstruct" the hardware interface, so that features are available individually, and can be used for additional purposes [23, 34].

6 Concluding Thoughts

While the discussion above spans much of the history of transactional memory, and mentions many open questions, the coverage has of necessity been spotty, and the choice of citations idiosyncratic. Many, many important topics and papers have been left out. For a much more comprehensive overview of the field, interested readers should consult the book-length treatise of Harris, Larus, and Rajwar [17]. A briefer overview can be found in chapter 9 of my synchronization monograph [31].

My sincere thanks to Hagit Attiya, Shlomi Dolev, Rachid Guerraoui, and Nir Shavit for organizing the celebration of Maurice's 60th birthday, and for giving me the opportunity to participate. My thanks, as well, to Panagiota Fatourou and Jennifer Welch for arranging the subsequent write-ups for BEATCS and SIGACT News. Most of all, my thanks and admiration to Maurice Herlihy for his seminal contributions, not only to transactional memory, but to nonblocking algorithms, topological analysis, and so many other aspects of parallel and distributed computing.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich, editors. Draft Specification of Transaction Language Constructs for C++. Version 1.1, IBM, Intel, and Sun Microsystems, Feb. 2012.

[2] Y. Afek, H. Avni, and N. Shavit. Towards Consistency Oblivious Programming. In *Proc. of the 15th Intl. Conf. on Principles of Distributed Systems*, pages 65-79. Toulouse, France, Dec. 2011.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-Free Made Fast. In *Proc. of the 27th ACM Symp. on Theory of Computing*, 1995.

[4] K. Agrawal, J. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[5] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, Velen, Germany, June–July 1993.

[6] H. W. Cain, B. Frey, D. Williams, M. M. Michael, C. May, and H. Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Proc. of the 40th Intl. Symp. on Computer Architecture*, Tel Aviv, Israel, June 2013.

[7] C. Click Jr. And now some Hardware Transactional Memory comments. Author's Blog, Azul Systems, Feb. 2009. `blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html`.

[8] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010. Earlier but expanded version available as TR 959, Dept. of Computer Science, Univ. of Rochester, July 2010.

[9] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

[11] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proc. of the 30th ACM Symp. on Principles of Distributed Computing*, San Jose, CA, June 2011.

[12] C. Ferri, A. Viescas, T. Moreshet, I. Bahar, and M. Herlihy. Energy Implications of Transactional Memory for Embedded Architectures. In *Wkshp. on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods (EPHAM)*, Boston, MA, Apr. 2008. In conjunction with *CGO*.

[13] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, Aug. 2005.

[15] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[16] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[17] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, second edition, 2010.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[19] M. Herlihy. The Transactional Manifesto: Software Engineering and Non-blocking Synchronization. In *Invited keynote address, SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[20] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[21] M. Herlihy and Y. Lev. tm_db: A Generic Debugging Library for Transactional Programs. In *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Raleigh, NC, Sept. 2009.

[22] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[23] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. Technical Report 1594, Dept. of Computer Sciences, Univ. of Wisconsin–Madison, June 2007.

[24] A. Israeli and L. Rappoport. Disjoint-Access Parallel Implementations of Strong Shared Memory Primitives. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.

[25] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[26] A. Matveev and N. Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures*, Montreal, PQ, Canada, July 2013.

[27] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[28] M. Moir. Transparent Support for Wait-Free Transactions. In *Proc. of the 11th Intl. Wkshp. on Distributed Algorithms*, 1997.

[29] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005.

[31] M. L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, 2013.

[32] M. L. Scott. Transactional Semantics with Zombies. In *Invited keynote address, 6th Wkshp. on the Theory of Transactional Memory*, Paris, France, July 2014.

[33] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99-116, Feb. 1997. Originally presented at the *14th ACM Symp. on Principles of Distributed Computing*, Aug. 1995.

[34] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation Tradeoffs in the Design of Flexible Transactional Memory Support. *Journal of Parallel and Distributed Computing*, 70(10):1068-1084, Oct. 2010.

[35] J. Turek, D. Shasha, and S. Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, Vancouver, BC, Canada, Aug. 1992.

[36] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[37] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[38] L. Xiang and M. L. Scott. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *Proc. of the 27th ACM Symp. on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[39] L. Xiang and M. L. Scott. Software Partitioning of Hardware Transactions. In *Proc. of the 20th PPoPP*, San Francisco, CA, Feb. 2015.

[40] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization. In x. f. H.-P. Computing, editor, *Proc., SC2013: High Performance Computing, Networking, Storage and Analysis*, pages 1-11. Denver, Colorado, Nov. 2013.