Metastability Tolerant Computing

Ghaith Tarawneh,* Matthias Függer,† Christoph Lenzen‡
* School of Electrical and Electronic Engineering, Newcastle University, UK
† LSV, CNRS & ENS, Paris-Saclay, France
‡ Max Planck Institute for Informatics, Saarland Informatics Campus, Germany
ghaith.tarawneh@ncl.ac.uk — mfuegger@lsv.fr — clenzen@mpi-inf.mpg.de

***Abstract*—Synchronization using flip-flop chains imposes a latency of a few clock cycles when transferring data and control signals between clock domains. We propose a design scheme that avoids this latency by performing synchronization as part of state/data computations while guaranteeing that metastability is contained and its effects tolerated (with an acceptable failure probability). We present a theoretical framework for modeling synchronous state machines in the presence of metastability and use it to prove properties that guarantee some form of reliability. Specifically, we show that the inevitable state/data corruption resulting from propagating metastable states can be confined to a subset of computations. Applications that can tolerate certain failures can exploit this property to leverage low-latency and quasi-reliable operation simultaneously. We demonstrate the approach by designing a Network-on-Chip router with zero-latency asynchronous ports and show via simulation that it outperforms a variant with two flip-flop synchronizers at a negligible cost in packet transfer reliability.**

I. Introduction

Transferring data and control signals reliably across clock domain boundaries is a classic problem highly relevant to today's systems with tens to hundreds of clock and voltage domains. A signal generated in one clock domain and latched in another can violate the setup/hold time conditions of its destination flip-flop and drive it into a metastable state. Following flip-flops that sample the metastable flip-flop's output may then latch different values and/or become metastable themselves [1], compromising state transitions and data integrity in ways that are difficult to predict [2], [3].

The conventional approach to mitigating metastability is to use a chain of flip-flops known as a synchronizer to resample the crossover signal [4]. This grants the first flip-flop at the receiver side some time to resolve any metastable state before using its value in state/data computations. Although the probability that metastability propagates beyond the synchronizer remains non-zero, it is reduced by orders of magnitude. The reliability of this process is typically expressed in terms of the Mean Time Between Failures (MTBF):

$$\text{MTBF} = \frac{e^{t_s/\tau}}{f_c f_d T_w}\,, \tag{1}$$

where t_s is resolution time, τ is the metastability regeneration time constant, f_c is the clock frequency, f_d is the crossover signal transition rate and T_w is a reference time window. The resolution time t_s is the propagation delay slack of synchronizer flip-flops (approximately an integer multiple of the clock cycle). Two flip-flop synchronizers are most common and typically guarantee an MTBF of thousands of years (assuming $\tau = 20\,\text{ps}$, $f_c = f_d = 1\,\text{GHz}$ and $T_w = 1\,\text{ns}$). This reliability comes at a cost, however, as crossover signal transitions must propagate through the synchronizer before being used in state/data computations and so a latency equal to the synchronizer depth is incurred.

A. Main Idea

We propose a methodology to design synchronous components that perform synchronization as part of their state/data computations (and thus avoid synchronization latency) at the expense of predictable and tolerable errors. Central to our proposal is the idea that, even though the propagation of metastability is fundamentally impossible to prevent, metastability can be *contained* and its effects *tolerated* (with an acceptable failure probability). Containment means that the output ports of a component are always stable, even if its flip-flops may become metastable,[1] while tolerance is the ability to meet a certain functional specification despite the occasional manifestation of metastability. The conventional two flip-flop synchronizer can be thought of as the simplest metastability-containing component; its output is stable and it meets the functional requirement of copying a value in few clock cycles. Our scheme generalizes these properties to useful forms of computation, removing the need for dedicated synchronizers.

B. Contributions

The contributions of this paper are as follows. (i) We present a method to avoid synchronization latency, yet maintain desired reliability guarantees when transferring data and control signals across clock domain boundaries (Section II). (ii) We then develop a generic framework, in which we formalize our method and verify that the desired behavior is achieved (Sections III and IV). (iii) To demonstrate that the method has real practical relevance, we apply it to a Network-on-Chip (NOC) router with store-and-forward flow control (Section V). (iv) We finally simulate a NOC composed of 4×4 such routers, embedded in several clock domains, and compare its performance and reliability to a corresponding implementation based on routers with two flip-flop synchronizers. The comparison demonstrates that the method confers performance gains at an insignificant reliability cost in at least one target application.

[1]For brevity, we will omit further references to failure probability keeping in mind that it is implicit in the definition of *containment*.

II. Proposed Scheme

We divide the state graph of a synchronous design into *safe* and *unsafe* regions. The unsafe region contains states corresponding to computations triggered by the transition of a crossover signal or shortly thereafter (e.g. the latching and processing of an incoming data item). It has an ideal specification that applies in the nominal case. The occasional onset and propagation of metastability can cause invalid state transitions, but state logic and encoding are designed such that these transitions cannot leave the unsafe region until any metastable values have been resolved (with an acceptable failure probability). Unsafe states are therefore synchronizing states in which metastability is granted time to resolve. Once a safe state is reached, synchronization is complete and state transitions continue as per usual.

We guarantee containment by disallowing changes to the design's output ports in unsafe states, thus preventing metastability from being observed externally. At the same time, we attempt certain computations in unsafe states while keeping computations that are critical to the correct behavior of the component to safe states to isolate them from the impact of metastability. Metastability may therefore cause occasional invalid state transitions and data corruption, but the effects of these events are confined to a subset of computations whose corruption can be tolerated and does not result in catastrophic failures for the target application. The above properties are formalized and proven in Sections III and IV, and later demonstrated practically in Section V. For now, we continue building the intuition behind the scheme.

A. General Setup

We assume that the control circuit of a design is specified by a synchronous state machine that communicates with a sender outside its own clock domain using `req` and `ack` signals. The arrival of `req` triggers the machine into a series of state transitions, ending with acknowledging the sender and returning to an idle state. Unlike conventional synchronization, we do not protect the state machine by piping `req` through a flip-flop chain. Instead, `req` (and any accompanying handshake data bits) are used in state computations directly. In consequence, state flip-flops may occasionally become metastable.

As the state machine undergoes transitions following the arrival of `req`, it triggers computations which we refer to as *operations*. Examples of operations include (i) asserting a write signal to store the incoming data item in a register, (ii) acknowledging the sender and (iii) triggering a neighboring state machine into processing a shared memory buffer. Operations are triggered by signals that are decoded from the state register. In addition, we group the state machine and a neighboring part of its clock domain (typically a datapath) into a *component* and require that component output signals are always stable (i.e., operations that change component outputs can be performed in safe states only). Components are therefore state and data containers that serve as a hierarchical boundary for containing metastability, akin to synchronizers. Figure 1 depicts the general setup.

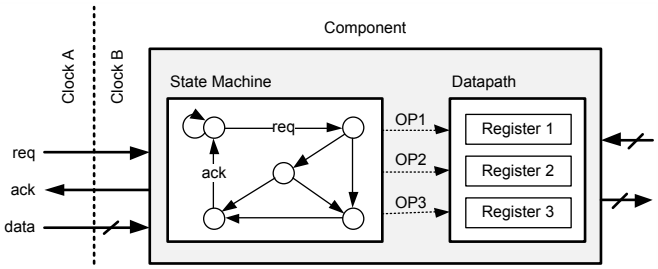

Figure 1. Metastability-containing component consisting of a state machine and associated registers (a datapath). State-decoded operation signals (OP_i) are used to trigger datapath computations. The arrival of a handshake request `req` triggers a series of state transitions and associated operations.

B. Quasi-reliability

Using conventional two flip-flop synchronizers can be viewed as a conservative special case of this scheme in which all computations are performed in safe states (*all safe*). Conversely, omitting synchronizers in a generic design may correspond to performing many/all computations in unsafe states (*all unsafe*). The proposed scheme permits operating between these two extremes, delineating computations based on their importance and leveraging the benefits of low-latency and quasi-reliable operation simultaneously.

III. Formal Model

We model the propagation of metastability in state combinational logic using a worst-case fashion as in [5]. We assume that a constant number $k \geq 1$ of clock cycles (typically one or two) is deemed sufficient to guarantee that any metastable state propagating in the state register has been resolved with a sufficiently high probability. Further, to prevent metastability from propagating beyond the component, we disallow operations that cause changes to component outputs within the first k clock cycles following the transition of `req`.

We will refer to (the control logic of) a component that receives data items from an asynchronous sender using handshake signals `req` and `ack` and has additional input/output signals to communicate with its synchronous neighbors. We demonstrate our scheme by discussing an implementation of this component that does *not* pipeline the sender's `req` signal through a synchronizer chain. We show how to guarantee that its state machine resolves any metastable states (with an acceptable MTBF) before making any changes to output ports (i.e. the component is metastability-containing). Moreover, we ensure that any lower bound on the time from being externally triggered until asserting `ack` (or modifying other output signals) is satisfied, even if the state machine becomes metastable. This ensures that (i) handshake protocol requirements are met and (ii) metastability cannot propagate to subsequent synchronous logic. We give a concrete example of this approach later, in Section V; in this section, we discuss how the behavior of the state machine under metastability is modeled and explain a general transformation ensuring that the above properties are satisfied.

A. State Machine Model

Consider a clocked state machine that is initialized in a predetermined state, ready for receiving data. The machine's state is recorded in a b-bit *state register,* where the encoding of a state $s \in S$ is given by the b-bit string $\varepsilon(s)$. On each clock cycle of the state machine, the state (i.e. the content of the state register) is updated based on (i) the current state, (ii) the value of `req`, and (iii) the state of an associated *datapath register*. Moreover, the state machine may decide to modify the datapath register or trigger output port changes via operations that are associated with certain states.

The behavior of the state machine is subject to the following constraints:

(1) The initial state is left when the (sampled) `req` signal indicates that new data has arrived; not before.
(2) Triggered operations that modify output ports must occur in the *safe region,* the set of states that cannot be reached before possible metastability has been resolved, i.e. k clock cycles passed since the transition of `req`.
(3) The datapath register is split into *control* and *data* parts. The *control part*
   - is only modified in safe states,
   - may affect the transitions of the state machine, and
   - may be used in operations,

   while the *data part*
   - may be modified in any state, but
   - must not affect the transitions of the state machine, and
   - cannot be used in operations.
(4) The state machine is guaranteed to reach a safe state and then transition back to the initial state. We assume that this includes completing the handshake by asserting `ack`.
(5) When this occurs, the data part of the datapath register must have been processed correctly if no metastable state was induced by `req`. Here, "processing" includes modifications performed as a result of state machine operations. Note that the precise meaning of "correctly processed" is application-specific.
(6) The control part of the datapath register must be processed correctly even if metastability was induced.

Thus, we allow non-critical modifications to be performed in the unsafe region. This permits saving synchronization time by performing computations optimistically at the expense of a small risk of data corruption.

B. Behavior under Metastability

In order to clarify the behavior of the state machine when facing metastability, we make use of the framework in [5]. Rather than specifying the effects of metastability directly, we define the transition function for metastability-free states and use the classification results from [5] to derive the transitions under metastability.

Recall that state transitions are based on (i) the sampled `req` signal, (ii) the current state (b bits wide), and (iii) the control part of the datapath register (c bits wide). Denote by $f : \{0,1\}^{1+b+c} \to \{0,1\}^{b+c}$ the transition function of the state machine (we include possible operations on the control part). Along the lines of [5], we next extend the specification of the transition function f to potentially metastable bits: in addition to the stable values 0 and 1, a signal may also attain the value M. Then the following definition describes the best possible guarantee we can make for inputs from $\{0,1,\mathrm{M}\}^{1+b+c}$.

Definition 1 (Metastable Closure): For bit strings $x, y \in \{0,1,\mathrm{M}\}^n$, define $x * y$ via

$$\forall i\colon (x * y)_i = \begin{cases} x_i & \text{if } x_i = y_i \\ \mathrm{M} & \text{else}\,. \end{cases}$$

Moreover, for $x \in \{0,1,\mathrm{M}\}^n$, set

$$\mathrm{Res}(x) = \{x' \in \{0,1\}^n \mid \forall i\colon x'_i \neq x_i \Rightarrow x_i = \mathrm{M}\}\,,$$

i.e. the set of strings that could result from resolving metastability in x. Then, for $g : \{0,1\}^n \to \{0,1\}^m$, its metastable closure $[g] : \{0,1,\mathrm{M}\}^n \to \{0,1,\mathrm{M}\}^m$ is defined by

$$\forall i\colon [g]_i(x) = \mathop{*}_{x' \in \mathrm{Res}(x)} g_i(x')\,.$$

As shown in [5], one can always design a circuit such that, under worst-case propagation of metastability, the circuit computes $[f](x)$ given input $x \in \{0,1,\mathrm{M}\}^{1+b+c}$, and that no (combinational) circuit can make more restrictive guarantees on its output. Accordingly, in the following we assume that the state machine is described by a binary state transition function f and is extended to metastable inputs by taking $[f]$. We stress that the encoding of states plays an important role: Intuitively, metastable "state" s encoded by $\varepsilon(s) \in \{0,1,\mathrm{M}\}^b$ is a superposition of all states s' such that $\varepsilon(s') \in \mathrm{Res}(\varepsilon(s))$.

Denote by $\mathtt{req}(r)$, $r \geq 0$, the state of the `req` signal *right before* sampled by clock edge $r+1$. Then, `req` (the string of samples) contains either $011\ldots$ or $0\mathrm{M}11\ldots$: per handshake-cycle, there can be at most one clock transition that samples a metastable `req`, since `req` is externally held at 1 until `ack` is raised by the state machine, completing the handshake cycle.

Recall that we require that the state machine stays in the (unsafe) initial state while $\mathtt{req}(r) = 0$, in which it does not modify the control part of the datapath register. Thus, for any $z \in \{0,1\}^{b+c}$, we have that $f(0 \circ z) = [f](0 \circ z) = z$; here $\circ$ denotes concatenation of strings. In the analysis, we can therefore assume that `req` is either $\mathrm{M}11\ldots$ or $11\ldots$

Given a Boolean transition function f satisfying the above, we are now ready to specify the behavior of the state machine. Denote by $\varepsilon(0) \in \{0,1\}^b$ the encoding of the starting state and let $x(0) \in \{0,1\}^c$ be any valid control part. At clock edge r, $r \in \{1,\ldots,k\}$, the state is updated (latched) as

$$y(r) = \varepsilon(r) \circ x(r) = [f](\mathtt{req}(r-1) \circ y(r-1))\,.$$

At clock edge $r = k+1$, the computation is the same, but we assume that metastability is resolved (with sufficiently high probability) after k cycles. Accordingly, let

$$y(k+1) = \mathrm{Res}([f](\mathtt{req}(k-1) \circ y(k-1)))\,.$$

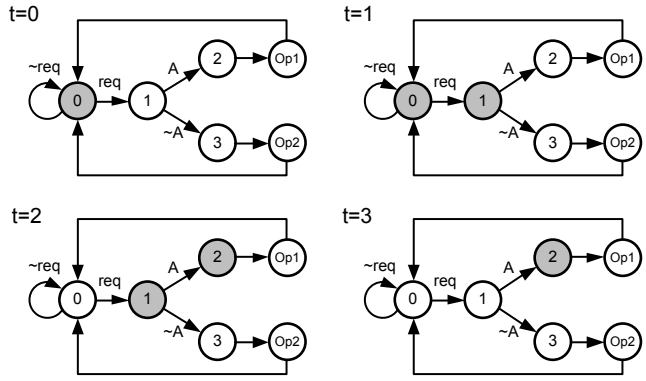

Figure 2. State diagrams illustrating an example execution of a state machine. Current states are gray and t is the index of the clock edge after the transition of req. State superpositions are indicated by multiple current states, here occuring during the first and second clock cycles. Once $t = 3$ is reached, metastability is resolved (assuming $k = 2$).

Afterwards, the state is binary, and the state machine follows its binary specification, i.e., for all $r \geq k + 2$,

$$y(r) = f(\texttt{req}(r-1) \circ y(r-1)) \,.$$

C. Example

Consider the state machine whose execution states are shown in Figure 2. Here, the datapath control part is a single bit A that is used to trigger one of two operations, Op1 or Op2. We assume that both operations involve updates to certain registers which we wish to guard against corruption. Op1 and Op2 must therefore not be triggered while the state register is metastable. The figure shows an execution of the machine in case $\texttt{req} = \mathrm{M}11\ldots$ and $A = 1$ (data transformation and state encoding rules are not included in the figure). We assume that metastability is guaranteed to resolve within $k = 2$ clock cycles, i.e. by the third clock edge after the transition of req. In this example, we trace a very deep metastable state that persists up to this limit. The machine behaves as follows. $\texttt{req}(0) = \mathrm{M}$ results in a metastable state at clock edge 1. From $\texttt{req}(1) = 1$, we have that the state machine is guaranteed to have left the initial state at clock edge 2. From $k = 2$, we have that metastability is guaranteed to resolve by the subsequent clock edge, here resulting in (metastability-free) state 2 at clock edge 3. Afterwards, the machine transitions through Op1 and then finally back to 0 (acknowledging the sender). Our requirement that neither Op1 or Op2 is triggered during metastability is therefore met. Observe that the metastable req only resulted in a delay of one clock cycle before triggering Op1, when compared to a circuit where req is guaranteed to always be sampled as either 0 or 1. Even better, $\texttt{req}(0) = \mathrm{M}$ means that req was just transitioning from 0 to 1 when the signal is sampled. This implies that the delay in terms of wall-clock time is just marginally larger than it would be if the signal transition occurred right after the signal is sampled.

D. Single Point of Failure

The reliable transfer of data between clock domains requires using the handshake request signal to control the passage of data (i.e. having a single point of synchronization [2]). This is satisfied in our scheme since all datapath operations (including the latching of incoming data) are triggered by state transitions that follow the handshake request. We also satisfy the bundled data constraint in the same way as conventional synchronization, i.e. by constraining crossover path delays such that req transitions arrive later than their accompanying data transitions by a sufficient margin (factoring in any combinational logic in crossover paths) [6]. We include in our definition of "data transitions" the transitions of req that arrive at datapath flip-flops, to prevent data corruption on the clock edge where req is sampled by the state register [7]. Note that incoming data bits are guaranteed to be stable (by the bundled data constraint) and can therefore be treated as part of the datapath register's control part (i.e. used to control state transitions and trigger operations in both safe and unsafe states).

E. Failure Types and Rates

We make a distinction between two forms of failures that may occur under our scheme. First, a *synchronization failure* is an event in which metastability reaches the safe region of the state graph, akin to propagating beyond a synchronizer chain. This is a failure that we are interested in making very improbable (i.e. with a MTBF in the order of thousands of years). For a conservative upper bound on the probability of this failure, we use Equation (1) assuming $t_s = k \times t_{ps}$ where t_{ps} is the worst-case propagation delay slack in state combinational logic. The resulting MTBF is less than that of a flip-flop chain synchronizer (that provides k clock cycles of synchronization time), but still likely to be large: given that clock periods are typically constrained by data critical paths, one can expect t_{ps} to be a large fraction of the clock period. In any case, different values of k can be chosen to achieve arbitrary levels of reliability.

The second type are *datapath failures*. These are more frequent events in which metastability corrupts register values that are updated in unsafe states. Datapath failures are undesired but predictable events that we here treat as a performance issue.[2] While we are interested in making synchronization failures very improbable, datapath failures need only be infrequent enough to not affect performance significantly. It is therefore more informative to express the frequency of these events as a probability per handshake P, which we calculate as

$$P = T_w f_c e^{-t_s/\tau} \,, \tag{2}$$

this time substituting t_s as the worst-case propagation delay slack in all combinational paths, not just state logic. Synchronization MTBF and datapath failure rates for an actual design are presented in Subsection V-E.

[2]Note that the consequences of datapath failures are application-specific and may not be limited to performance. However, these consequences are predictable and can therefore be addressed by the component's environment. We discuss this further in Subsection VI-C.

IV. State Encoding

In the example in Figure 2, we did not discuss state encoding; we implicitly assumed an encoding that is optimal with respect to uncertainty. In contrast, choosing a "bad" encoding can have severe consequences as detailed below.

Consider any encoding for which $\varepsilon(0) = 000$, $\varepsilon(1) = 111$, and all other states are encoded with pairwise distinct values in $\{0,1\}^3$. In this case, $\mathtt{req}(0) = \mathrm{M}$ makes the system go to "state" MMM at time 1: a superposition of all states. In terms of Figure 2, this means that all states are marked gray at time 1. With this encoding, the unsafe region is thus the set of all states and operations Op1 and Op2 can be triggered from unsafe states, contrary to our requirements in the example.

A much better choice is the encoding $\varepsilon(0) = 000$, $\varepsilon(1) = 001$, $\varepsilon(2) = 011$, $\varepsilon(3) = 101$, $\varepsilon(\text{Op1}) = 111$, and $\varepsilon(\text{Op2}) = 110$. Then $\mathtt{req}(0) = \mathrm{M}$ results in "state" 00M at time 1: a superposition of states 0 and 1 like in Figure 2. If $A = 0$, the system transitions to M01 (superposition of 1 and 3), and if $A = 1$ to 0M1 (superposition of 1 and 2) at time 2. This is the same behavior as in the example of Figure 2. From the fact that metastability must decay at time 3 at the latest, we easily check that all following transitions (including those to Op1 and Op2) are metastability-free. It follows that the requirements (1) to (4) specified earlier are fulfilled.

A. Finding Good Encodings

While the encoding before was handcrafted, the question arises if there is a systematic way to derive small bit-size encodings that allow to restrict unsafe regions. To address this problem, we first capture what it means for two state machines (on different encodings) to behave the same way with respect to the underlying Boolean specification f.

Definition 2 (State Graph): For an encoding $\varepsilon(s)$, $s \in S$, and a transition function f, the directed *state graph* $G = (S, E)$ is defined by

$$(s,t) \in E \Leftrightarrow \exists x, x' \colon f(1 \circ \varepsilon(s) \circ x) = \varepsilon(t) \circ x' \,.$$

Definition 3 (Equivalent State Machines): Given states S, encodings ε and ε' of b and b' bits, and transition functions $f \colon \{0,1\}^{1+b+c} \to \{0,1\}^{b+c}$ and $f' \colon \{0,1\}^{1+b'+c} \to \{0,1\}^{b'+c}$, respectively, we consider the corresponding state machines *equivalent* iff

- f and f' have the same state graph and
- for each edge (s,t) of the state graph and $x \in \{0,1\}^c$, there is $x' \in \{0,1\}^c$ with $f(1 \circ \varepsilon(s) \circ x) = \varepsilon(t) \circ x'$ and $f'(1 \circ \varepsilon'(s) \circ x) = \varepsilon'(t) \circ x'$.

Note that equivalent state machines do *not* necessarily behave the same way in the presence of metastability, but their behavior on non-metastable states is identical.

The following result states that we can always change the encoding so that (i) the number of required bits does not increase too much, (ii) the new and the old state machine are equivalent, and (iii) the new encoding guarantees that the unsafe region consists of the states within k directed hops of the starting state.

Theorem 4: Given any b-bit encoding $\varepsilon(s)$ for $s \in S$ and transition function $f \colon \{0,1\}^{1+b+c} \to \{0,1\}^{b+c}$, we can find a b'-bit encoding $\varepsilon'(s)$ and a transition function $f' \colon \{0,1\}^{1+b'+c} \to \{0,1\}^{b'+c}$ such that

- $b' = b + k + 1$,
- the respective state machines are equivalent, and
- the unsafe region for f' is a subset of the states that are in (directed) distance at most k from the starting state in the state graph.

Proof: We only give the proof idea here. We modify the state encoding and transitions as follows. In addition to the encoding that is already given, we reserve $k+1$ additional bits for unary encoding of the distance to the starting state, where states in distance $k+1$ or larger simply have all bits set to 1. The new transition function f' consists of applying f to the input, ignoring the $k+1$ reserved bits, and operating on them as described above. We then show by induction that at clock edge $r \geq 1$, only the first r distance bits can be modified. ■

Given that k is no more than 1 or 2 in practice, this encoding scheme induces a very small overhead compared to the optimum of $\lceil \log |S| \rceil$ bits. From Theorem 4, we obtain:

Corollary 5: Given any state set S and transition function, we can find an encoding with $\lceil \log |S| \rceil + k + 1$ bits and an equivalent state function such that the unsafe region is a subset of the states that are in (directed) distance at most k from the starting state in the state graph.

We can also be more restrictive in terms of the unsafe region (as well as metastability causing "jumps" to otherwise unreachable states).

Theorem 6: Given any b-bit encoding $\varepsilon(s)$ for $s \in S$ and transition function $f \colon \{0,1\}^{1+b+c} \to \{0,1\}^{b+c}$, we can find a b'-bit encoding $\varepsilon'(s)$ and a transition function $f' \colon \{0,1\}^{1+b'+c} \to \{0,1\}^{b'+c}$ such that

- $b' = b + \sum_{i=0}^{k} \lceil \log(\Delta_i + 1) \rceil$, where Δ_i is the maximum outdegree of nodes within i hops from the starting state,
- the respective state machines are equivalent, and
- the unsafe region for f' is exactly the union over all valid control parts x of states that can be reached in at most k cycles from the starting state with control part x when $\mathtt{req}(0) = 1$.

Proof: We follow a similar approach as in Theorem 4. The difference is that instead of encoding merely that distance r is reached, we rather encode the edge over which the previous state was left. Concretely, for each state within distance k from the starting state, we enumerate its outgoing edges; at clock edge $r \leq k+1$, we store the (number of) the edge over which the previous state was left. To this end, we reserve $\log\lceil \Delta_i + 1 \rceil$ many bits, as we also require a codeword $\perp$ indicating that the r-th transition has not been performed yet. Again, f' acts like f on all but the additional bits, which are determined as just described.

Fixing any control part x, reasoning as for Theorem 4 shows that the safe region cannot be entered before clock edge $k+1$. By construction, f' with the new encoding is equivalent to f with the original one, and $b' = b + \sum_{i=0}^{k} \lceil \log(\Delta_i + 1) \rceil$. ■

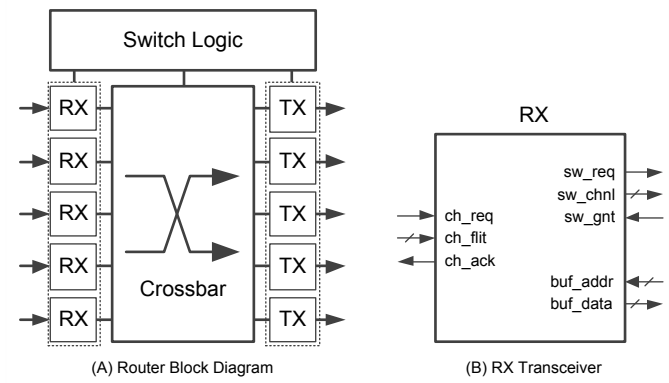

Figure 3. (A) Router: lines are bidirectional interfaces and arrows indicate direction of flit transfer. (B) RX Transceiver: arrows are individual signals/buses.

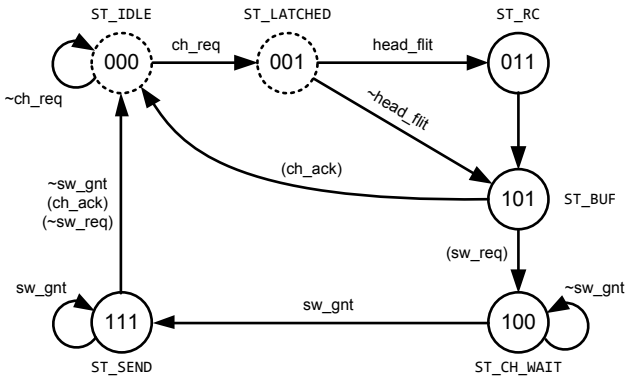

Figure 4. State diagram (RX transceiver). Edge labels denote transition conditions and, between brackets, output port changes (unlabeled transitions are unconditional). Unsafe states are marked with a dashed contour. Port changes are made only during safe states. The labels `ch_req` and `ch_ack` denote signal changes (two-phase handshake) while others signal levels.

V. Implementation

In this section, we present a NOC router that transports packets across clock domains and uses our method to remove synchronization latency in the forward direction. Our aim here is not to present an optimal or efficient router (router design and trade-offs are outside the scope of the current work). Instead, we focus on demonstrating our scheme using NOC routing as a target application.

A. Router Design

Figure 3A shows a block diagram of the router. The router has 5 channels (north, east, south, west and local), each comprising a pair of receive/send transceivers. The transceivers use two-phase handshakes to communicate with other transceivers in adjacent routers, each operating in its own clock domain. The router relies on store-and-forward flow control: packets are received and assembled, one flit at a time, before being retransmitted. Switch logic allocates access to outgoing channels using round-robin arbitration and XY routing is used.

B. The Transceiver

We used our scheme to avoid synchronizing incoming flits in the RX transceiver (henceforth *the transceiver*). This unit's block and state diagrams are shown in Figures 3B and 4 respectively. The transceiver serves as the metastability-containing component in the router (following the definition in Subsection II-A). In what follows, we describe its behavior and how it integrates with other router modules.

Initially, the transceiver is in an idle state (`ST_IDLE`). Upon receiving a handshake request (`ch_req`), the flit is latched and its control bits are examined. If the control bits denote a header flit (`head_flit` = 1), the transceiver extracts the packet destination identifier and fetches outgoing channel information from a routing table (routing computation, `ST_RC`), then appends the flit to the buffer (`ST_BUF`). Body flits (`head_flit` = 0) are appended to the buffer without prior routing computation. Once the packet is fully assembled (a packet comprises 8 flits), the transceiver issues a request for the outgoing channel (`ST_CH_WAIT`) and, when granted, waits for the packet to be fully transmitted (`ST_SEND`) before returning to the idle state.

The transceiver has three interfaces:

(1) Channel interface (asynchronous, two-phase handshake): receives incoming flits from remote sender
(2) Switch interface (synchronous): requests outgoing channel from switch logic and receives send notification
(3) Buffer interface (synchronous): provides packet buffer read access to outgoing channel TX transceiver

C. Metastability Containment

The transition of `req` triggers the transition from the idle state `ST_IDLE` and subsequent processing of the incoming flit. The set {`ST_IDLE`, `ST_LATCHED`} is the unsafe region, while remaining states constitute the safe region. Here, we assume that metastable states resolve within 1 clock cycle with an acceptable MTBF (i.e. $k = 1$, corresponding to a two flip-flop synchronizer). Note that this can easily be generalized to larger k. To satisfy the conditions for metastability containment, all output port changes are performed in safe states.

D. Datapath Failure Modes

Datapath failures are caused by the persistent corruption of registers that are write-accessible in the two unsafe states. These registers are:

(1) `REG_FLIT` (flit register): latches the flit received via the channel interface, written when entering `ST_LATCHED`;
(2) `REG_OUT_CHANNEL` (outgoing channel register): holds the fetched destination of the current packet, written when entering `ST_RC`;
(3) `REG_BUFF_WR_IND` (buffer write index): holds the index of the current flit, written when entering `ST_BUF`;
(4) `MEM_BUFF` (packet buffer): stores current packet, written when entering `ST_BUF`.

Datapath failure modes can be inferred as the worst-case consequences of committing arbitrary values to these registers:

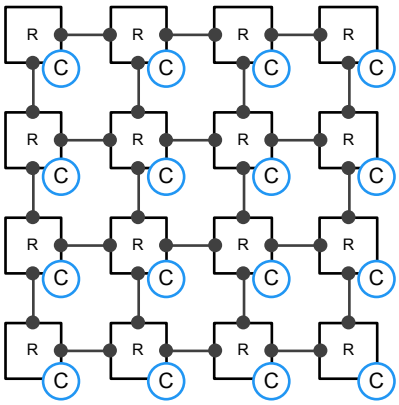

Figure 5. Architecture of simulated NOCs. Squares are routers and edges are bidirectional transceiver connections. Circles are dummy cores that generate packets with random destinations and consume incoming packets. Each core is connected to its adjacent router via the local channel.

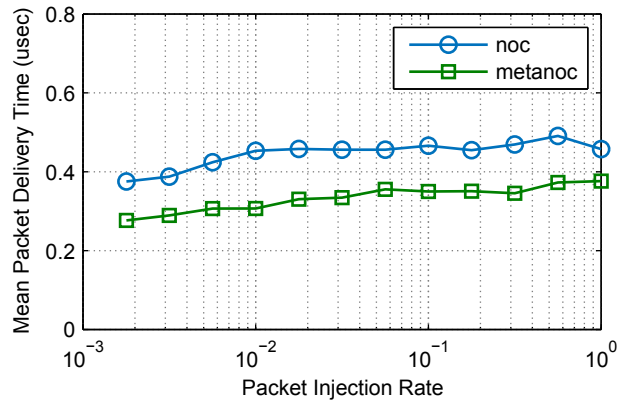

Figure 6. Simulation results showing 17.7 – 26.2% reductions in mean packet delivery time when using the proposed scheme. `noc` is a NOC with two flip-flop synchronizers while `metanoc` uses the presented zero-latency RX transceivers to contain metastability and tolerate its effects. Both NOCs consist of a 4×4 router mesh with local dummy cores. Both use XY routing, round-robin arbitration and store-and-forward flow control.

(1) assembly and transmission of corrupt packets;
(2) forwarding packets via incorrect outgoing channels;
(3) dropping packets (when the buffer index does not reach the packet flit count and is reset by the head flit of the subsequent packet).

E. Performance and Reliability

We implemented the proposed router in Verilog and compared its performance and reliability to a corresponding implementation based on conventional two flip-flop synchronizers. We simulated a 4×4 NOC (Figure 5) based on each router type and calculated mean packet delivery time for random traffic patterns. Clock frequency was 800 MHz for all routers (we assume independent plesiosynchronous clock sources). Figure 6 shows the results of our simulations. We observed an improvement of 17.7 to 26.2% in mean packet delivery time across a range of packet injection rates as a consequence of applying our scheme to the RX transceiver.

We synthesized the router using a commercial 65nm cell library, targeting a clock frequency of 800 MHz, to obtain realistic estimates of propagation delay slack and calculate synchronization MTBF and datapath failure rates. State combinational logic had a propagation delay slack t_{ps} = 0.95 ns. Assuming f_d = 800 MHz, T_w = 1 ns and τ = 20 ps, the MTBF of synchronization is 2.11×10^4 years. Note that while this is less than the MTBF of a two flip-flop synchronizer chain, a higher MTBF (if desired) can be obtained by increasing k.

The worst-case propagation delay in the synthesized transceiver (including state and datapath combinational logic) was 0.2 ns. Using Equation (2), we calculated a datapath failure probability per handshake of $P \leq 0.000036$. To determine the impact of datapath failures, we modified our simulation to write random values (with probability P) to all registers that are write-accessible in unsafe states (listed in Subsection V-D). We found that this caused packet loss at a rate of <0.1% and packet corruption at a rate of <0.02% but no significant difference in mean packet delivery time.

VI. Discussion

A. Related Work

Large part of the work in synchronization literature (i) presents special case solutions, e.g. for plesiosynchronous, rationally-related, and periodic clocks [8]–[10], or (ii) explores time-unbounded decision schemes such as pausible/stretchable clocking [11]–[13]. Few proposed "speculative" synchronization schemes do not target the clocking process itself as such but focus on overlapping synchronization with computation using architectural techniques [7], [14], [15]. The presented scheme is similar to speculation in this regard but is characteristically different. While speculation involves using a dedicated synchronization path to identify then correct/discard corrupt results once synchronization is complete, metastability containment *combines* synchronization and computation and makes no attempt to identify nor correct the resulting errors. Instead, the approach focuses on isolating metastability and predicting its consequences at the component level.

Several recent papers offer ideas that complement the work presented here. Of particular relevance are the metastability-containing circuits presented in [5], [16], [17] that eliminate/minimize the propagation of metastable operands when computing Boolean functions. In particular, the described behavior of state machines under metastability is achieved only when the circuit specification is implemented in a certain way, cf. [5]. These techniques can further constrain the effects of metastability when performing computations in the unsafe region and hence improve the overall reliability of our scheme; for example, under certain conditions it is possible to execute operations correctly *despite* internal metastability of the state machine. Finally, metastability modeling and verification frameworks [18] can also facilitate using our scheme by enabling designers to formally verify properties that describe metastability masking and propagation. This is particularly

useful for verifying that only specific registers may be written to and that module output interfaces are unchanged while in the unsafe region of the state graph.

B. Nominal Synchronization Performance

The synchronization time needed for a satisfactory MTBF (e.g. thousands of years or more) is much longer than the average resolution time of a metastable state. For example, assuming $\tau = 20\,\text{ps}$, $f_c = 1\,\text{GHz}$ and $T_w = 1\,\text{ns}$, the probability that a synchronization attempt causes a metastable state longer than 10% of the clock period is less than 1%. One may therefore differentiate between the *worst* and *nominal* cases of metastability resolution. While flip-flop chain synchronizers target the worst case, our scheme takes advantages of the nominal case by attempting computations optimistically then containing and tolerating the resulting (infrequent) failures. The presented router design is a case in point; a propagation delay slack of 16% of the clock period (0.2/1.25 ns) was sufficient to reduce packet corruption/drop rates below 0.1%.

C. Shifting the Problem of Metastability

The system-level consequences of component errors vary significantly across applications (and may be catastrophic in certain cases), so the proposed scheme is no general solution to the problem of synchronization. Nevertheless, shifting the reliability issues posed by metastability to higher abstraction layers confers three advantages. First, it permits exploiting any ability to tolerate reliability issues that is intrinsic to the application [19], [20]. Second, it enables the re-use of existing high-level reliability mechanisms that may be present to mitigate other forms of low-level errors [21]. Third, it exposes metastability failures to a richer set of high-level design tools, trade-offs and solutions than present at the circuit level. The latter is exemplified by NOC routing where (as we have shown) metastability can be transformed into a packet delivery reliability issue, a relatively well-studied problem for which several solutions have been proposed [22]–[24].

VII. Conclusion

Metastability is traditionally viewed as a malevolent circuit-level phenomenon that can cross hierarchical boundaries and cause catastrophic system-level failures. While it is true that the propagation of metastability is fundamentally impossible to prevent, this propagation is constrained structurally and functionally. Metastability can therefore be contained within a subset of state/data flip-flops and prevented from propagating further until its probability is sufficiently diminished. We presented a design scheme in which we exploited this idea to avoid dedicated synchronization while isolating metastability at the component level and predicting its consequences.

Acknowledgments

We thank Stephan Friedrichs and Andrey Mokhov for their feedback and the useful discussions. This work was supported by EPSRC grant EP/N031768/1 (project POETS) and by the Austrian Science Foundation (FWF) project SIC (P26436). Matthias Függer was with Max Planck Institute for Informatics at the time of writing.

References

[1] T. J. Gabara, G. J. Cyr, and C. E. Stroud, "Metastability of CMOS master/slave flip-flops," *Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 10, pp. 734–740, 1992.

[2] R. Ginosar, "Metastability and synchronizers: A tutorial," *Design and Test of Computers*, vol. 28, no. 5, pp. 23–35, 2011.

[3] S. Beer, M. Cannizzaro, J. Cortadella, R. Ginosar, and L. Lavagno, "Metastability in better-than-worst-case designs," in *20th Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2014, pp. 101–102.

[4] I. W. Jones, S. Yang, and M. Greenstreet, "Synchronizer behavior and analysis," in *15th Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2009, pp. 117–126.

[5] S. Friedrichs, M. Függer, and C. Lenzen, "Metastability-containing circuits," *CoRR*, vol. abs/1606.06570, 2016. [Online]. Available: http://arxiv.org/abs/1606.06570

[6] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design*. Springer, 2002.

[7] G. Tarawneh, A. Yakovlev, and T. Mak, "Eliminating synchronization latency using sequenced latching," *Transactions On Very Large Scale Integration (VlSI) Systems*, vol. 22, no. 2, pp. 408–419, 2014.

[8] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *9th Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2003, pp. 78–88.

[9] L. F. Sarmenta, G. A. Pratt, and S. A. Ward, "Rational clocking [digital systems design]," in *Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 1995, pp. 271–278.

[10] W. J. Dally and S. G. Tell, "The even/odd synchronizer: A fast, all-digital, periodic synchronizer," in *16th Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2010, pp. 75–84.

[11] J. Kessels, A. Peeters, P. Wielage, and S.-J. Kim, "Clock synchronization through handshake signalling," *Microprocessors and Microsystems*, vol. 27, no. 9, pp. 447–460, 2003.

[12] K. Y. Yun and A. E. Dooply, "Pausible clocking-based heterogeneous systems," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 4, pp. 482–488, 1999.

[13] R. Dobkin, R. Ginosar, and C. P. Sotiriou, "High rate data synchronization in gals socs," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 10, pp. 1063–1074, 2006.

[14] D. J. Kinniment and A. Yakovlev, "Low latency synchronization through speculation," in *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer, 2004, pp. 278–288.

[15] G. Tarawneh and A. Yakovlev, "An RTL method for hiding clock domain crossing latency," in *19th Conference on Electronics, Circuits and Systems (ICECS)*, 2012, pp. 540–543.

[16] C. Lenzen and M. Medina, "Efficient Metastability-Containing Gray Code 2-Sort," in *22nd Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2016.

[17] J. Bund, C. Lenzen, and M. Medina, "Near-Optimal Metastability-Containing Sorting Networks," in *Design, Automation and Test in Europe (DATE)*, 2017.

[18] G. Tarawneh, A. Mokhov, and A. Yakovlev, "Formal verification of clock domain crossing using gate-level models of metastable flip-flops," in *Design, Automation & Test in Europe (DATE)*, 2016, pp. 1060–1065.

[19] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *43rd Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[20] X. Lagorce, E. Stromatias, F. Galluppi, L. A. Plana, S.-C. Liu, S. B. Furber, and R. B. Benosman, "Breaking the millisecond barrier on spinnaker: implementing asynchronous event-based plastic models with microsecond resolution," *Frontiers in neuroscience*, vol. 9, 2015.

[21] M. De Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 497–508, 2010.

[22] Y. H. Kang, T.-J. Kwon, and J. Draper, "Fault-tolerant flow control in on-chip networks," in *4th Symposium on Networks-on-Chip (NOCS)*, 2010, pp. 79–86.

[23] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant network-on-chip architectures," in *Conference on Dependable Systems and Networks (DSN)*, 2006, pp. 93–104.

[24] H. Cho, L. Leem, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 546–558, 2012.