Latency-Insensitive Design and Central Repetitive Scheduling [*]

Julien Boucaron
INRIA
Sophia-Antipolis

Robert de Simone
INRIA
Sophia-Antipolis

Jean-Vivien Millo
INRIA
Sophia-Antipolis

Abstract

The theory of latency-insensitive design (LID) was recently invented to cope with the time closure *problem in otherwise synchronous circuits and programs. The idea is to allow the inception of arbitrarily fixed (integer) latencies for data/signals traveling along wires or communication media. Then mechanisms such as shell wrappers and relay-stations are introduced to "implement" the necessary back-pressure congestion control, so that data with shorter travel duration can safely await others with which they are to be consumed simultaneously by the same computing element. These mechanisms can themselves be efficiently represented as synchronous components in this global, asynchronously-spirited environment.*

Despite their efficient form, relay-stations and back-pressure mechanisms add complexity to a system whose behaviour is ultimately very repetitive. Indeed, the "slowest" data loops regulate the traffic and organize the traffic to their pace. This specific repetitive scheduling has been extensively studied in the past under the name of "Central Repetitive Problem", and results were established proving that so-called k-periodic optimal solutions could be achieved. But the "implementation" using typical synchronous circuit elements in the LID context was never worked out.

We deal with these issues here, using explicit representation of schedules as periodic words on $\{0,1\}^\star$ borrowed from the recently theory of N-synchronous *systems.*

1 Introduction

Today's SoC design faces the problem of *timing closure* and *clock synchronization*. Latency Insensitive Theory [14, 13, 12, 4] was introduced to tackle the timing closure issue. It does so by defining specific storage elements to install along the wires, which then divide signal and data transport according to necessary latencies. Dynamic scheduling schemes to avoid data congestion are implemented by additional signals. They slow down the faster traffic routes to the pace of others.

On the other hand, the general theory of weighted marked graph teaches us that there exist static repetitive scheduling for such computational behaviors [6, 1]. Such static k-periodic schedulings have been applied to software pipelining problems [10, 15], and later to SoC LID design in [5]. But these solutions pay in general little attention to the form of buffering elements that are holding values in the scheduled system, and their adequacy for hardware circuit representation. For instance in [5] the basic clock has to be divided further into multiphases, a solution which we find undesirable here. Also, in these solutions the precise activity allocation of clock cycles to computation nodes is not explicit (except in the restricted case of nested loop graphs).

Expressing an explicit precise static scheduling that uses predictable synchronous elements is desirable for a number of issues. It could easily be synthesized of course, but could also be evaluated for power consumption, by attempting to interleave as much as possible the computation phases with the transportation phases. It could also be used as a basis for the introduction of control modes with alternative choices that are currently absent of the model (it performs over and over the same computations in a partial order). We shall not consider these issue here, but they are looming over our approach to modeling static schedules.

Our contributions Our main objective is to provide a statically scheduled version of relaxed synchronous computation networks with mandatory latencies. It can be thought of as trying to add more latencies to the already imposed ones, in order to slow down tokens to exactly regulate the traffic. The new latencies are somehow virtual, as they are not uniquely defined and could be swallowed by more relaxed *redesign* of computation elements. The goal is to equalize all travel times (counted in clock cycles) in a relaxed synchronous firing rule, so that all flow-control back-pressure mechanisms can be done without, and relay-stations are simplified to mere single-slot registers. But

[*]Work supported by STMicroelectronics grants in the context of the french regional PACA CIM initiative and PS17 Project

sadly this is not always feasible, due to the fact that exact solutions would require sometime rational (rather than integer) delays to be inserted.

Our goal will be to introduce firstly "as many" integer latencies as possible, and then only deal with the extra fractional parts that may still resist prefect equalization. It will be done by inserting specific *fractional register* modeling elements. These should also be represented by hardware components, were relay-stations and shell wrappers. Preliminary attempts were conducted in [5], but they need to divide clock cycles into smaller phase, and we do not want this. We need to introduce elements that "capture" tokens to hold them less than each time. The pattern of hold-ups thus has to be made explicit.

In order to represent the explicit scheduling of computation node activities we borrow from the theory of N-*synchronous* processes [7], where such notions were introduced. We identify a number of interest in relative phenomena occurring when loops with rates that do not match (for instance involving prime numbers) are present. We shall also face the issues of the initialization phase, and of the recognition of the stationary cases.

Paper structure: In the next section we recall the *modeling framework* of computation nets and its semantics variations over the firing rules. We start with a recap of the now classical fully synchronous and fully asynchronous semantics. The introduction of weights (or *latencies* brings an intermediate model that lags in-between (retimed asynchronous or relaxed synchronous), with some slight distinctions remaining between them. We provide syntax to express explicit schedules, borrowed from the new theory of N-synchronous processes. It helps us phrase the problem of general equalization of latencies that we want to tackle. We end the section with a summary of important known results on k-periodic static schedules for Weighted Event graphs.

The following section describes our approach and the sequence of algorithmic steps of analysis required to introduce integer latencies, compute the schedules over the initialization and the stationary periodic phases of the various computation nodes, and identify locations where to add extra fractional latencies to even out fully the various data routes between computations. The goal is to maintain the throughput of the slowest loop cycle, which is the best attainable anyway. A formal synchronous description of the *Fractional Registers* involved is provided, with conditions on their application. Matters in efficient implementation are also mentioned (while we are currently building a tool prototype around these implementation ideas, it was not fully operational by submission time).

We provide a number of examples to highlights the current difficulties, and we end up with considerations on further topics.

2 The Modeling Framework

2.1 Computation nets

We shall loosely call *computation net scheme* a graph structure consisting of computation nodes, linked together by directed arcs. In such a simple model computations should consist in repeatedly consuming input values from incoming arcs, and producing output values on outgoing ones. There are also primary input and output arcs (without a source or target node respectively), and possibly loops in the graph. Behaviors do not in fact depend on actual data values, but only on their availability to computing nodes. They can be abstracted as presence token. The number of tokens in a loop stays invariant across computations.

The model is incomplete without a description of the firing rule, which enforces the precise semantics for triggering computations. The way token may (or may not) be stored at arcs between computations is an important part of these potential semantics. (Partial) computation orderings should enforce obvious properties: informally stated, production and consumption of data should ultimately match in rates. No data should be lost or overwritten by others, deadlocks (data missing) and congestions (data accumulated in excess) should be avoided altogether. Figure 1(a) displays a simple computation net scheme.

2.2 Synchronous, asynchronous and relaxed-synchronous semantics

There are two obvious starting points for possible semantics: the *fully synchronous* approach, in which all computations are performed at each cycle of a global clock,so that data are uniformly flowing; the *fully asynchronous* approach, with infinite buffers to store as many data as needed in between independent computations. The first approach is represented in the theory of synchronous circuits and synchronous reactive languages [2], the second in the theory of *Event/Marked graphs*[8]. Both lead to a profusion of theoretical developments in the past. In both case there is a simple correctness assumption to guarantee safety and liveness: the existence of (at least) a unit delay element in each loop in the synchronous case, the existence of (at least) a token in each loop in the asynchronous case. Recall that the number of data cannot blow up inside loops in the asynchronous case, as it remains invariant.

Fully synchronous and *fully asynchronous* semantics are somehow extreme endpoints on the semantic constraint scale. As a middle point proposals have been made to introduce explicit *latencies* on arcs to represent mandatory travel. In such a simple model what matters most is the potential time for data/tokens. Starting from the asynchronous side, this lead to the theory of *Weighted Event/Marked*

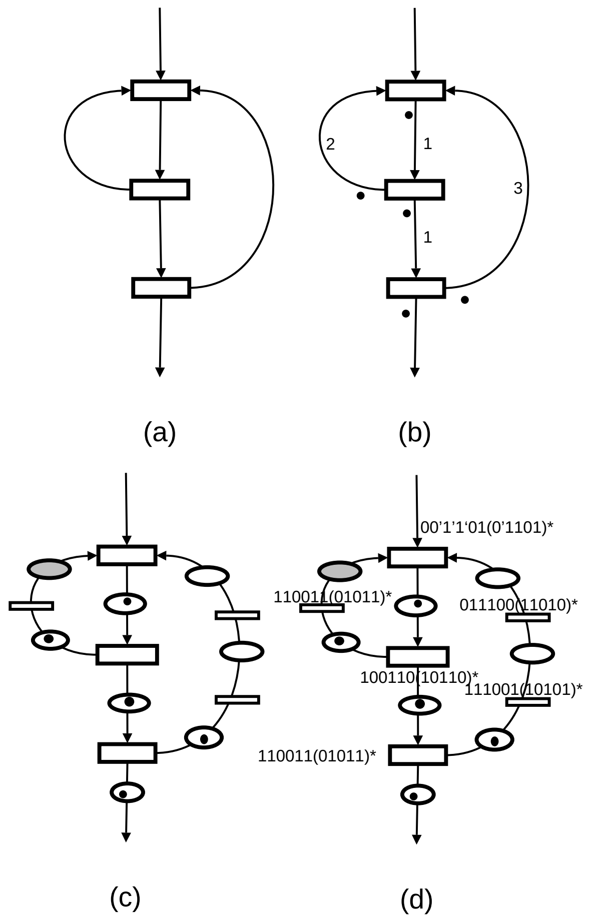

Figure 1. (a) An example of computation net, (b) with token marking and latency features, (c) with relay-stations dividing arcs according to latencies, (d) with explicit schedules

Graphs (WEM graphs), extensively studied in the past [1, 6] to model production systems. Starting from the synchronous side, this led to the theory of *Latency-Insensitive Design* (LID) [4], which attempts to solve the so-called *Timing Closure* issue in circuits where electric wires can no longer be assumed to propagate in unit time.

Figure 1(b) displays a computation network annotated with latencies. Figure 1(c) refines this description by making explicit the successive stages in the travel and the places where tokens may reside while on travel (the smaller rectangles can be thought of as *motion nodes*, similar to the computation nodes as they consume and produce token from one travel section to the next). For the sake of simplicity we shall assume for now that all places following a computation nodes are marked by a token (and only them), as shown on figure 1(c). Weaker assumptions can be taken, reflecting only the needs expressed previously (at least one token on every loop cycle).

Definition 1 (Rates and critical cycles). Let G be a Weighted Marked graph, and C a cycle in this graph.
The *rate* r of the cycle is equal to $\frac{T}{L}$, where T is the number of tokens in the loop (which is constant), and L is the sum of latencies labeling its arcs.
The *throughput* of the graph is defined as the min of rates over all cycles.
A cycle is called *critical* if its rate is equal (i.e, as slow) as the graph throughput.

2.3 Dynamic and static scheduling

Studies on both LID networks and WEM graphs both attempt at providing synchronous execution rules to the computation nets, in the sense that all computation nodes fire as soon as possible, possibly simultaneously. Their setting and motivations still differ in several ways:

- In LID theory [13, 12] the buffering places are replaced by so-called *relay-stations*, and the computation nodes are surrounded by *shell wrappers*. The purpose of these components, which are described as additional *synchronous* elements [3], is to implement a dynamic on-line scheduling scheme: it regulates data traffic to the point that never more than two tokens accumulate in any relay-station. Computations require all input tokens, but also free space in output relay-stations. In the example of figure 1 (c), in the second step of the initialization phase the relay-station in the grey place would need to hold two token values (while the token on the right arc travels up).

- Research conducted on WEM graphs attempts to obtain static repetitive scheduling, based on the fact that a synchronous firing of computation nodes leads to a deterministic behavior, which is bound to repeat itself with a given period because of the finiteness of possible token distributions. But the scheduling does not pay much attention to the distribution of token in between places (and thus the buffer sizes). The foundations of the theory of static and k-periodic scheduling for Weighted Marked Graphs is to be found in [6, 1]. In [6] the authors named it as *Central Repetitive Problem (CRP).*

In fact, it can easily be seen that the dynamic scheduling governing the synchronous firing rules of both LID and WEM lead to*ultimately repetitive* behaviors, thus amenable to static scheduling whenever they can be computed offline. Indeed, the lack of control in the (choice-free) systems ensures determinism. Each instantaneous configuration leads to a unique next one by firing simultaneously all possible computation and transportation nodes (here we call configuration an allocation of tokens to the buffers or relay-stations); the set of possible configurations is finite (remember that we assume our nets are strongly-connected graphs, which guarantees an invariant token numbers on each loop); then the system is bound to retrieve an already visited state after a finite initialization sequence, and the behavior will be identical from now on than the first sequence past this state.

So, a synchronous LID or WEM system behaviour consists of a transitory initialization part, followed by a stationary periodic one. Importantly, the same period length applies to all the computation nodes of the system, and inside the period the number of firing is also the same for all nodes (this is called the periodicity); So one can talk of the period and the periodicity of the graphs. See [6, 1] for details in the case of WEM graphs.

Note that in general there is no guarantee that the actual computation rate will match the one indicated by the local user-provided latencies. Data traveling on non-critical paths may have to wait for others on slower routes to join at computation nodes. In that sense the latencies indicated form a floor value to the effective ones, at least on non-critical paths.

[1] provides bounds for the period and periodicity sizes A bound for periodicity k is established as the periodicity of G^c, where G^c represents the restriction of G to its critical cycles. The periodicity of G^c is equals to the lcm (Least Common Multiple) of the periodicity of each SCC (Strongly Connected Component) of G^c. The periodicity of a SCC of G^c is equals to the gcd (Greatest Common Divisor) of the token count of all its cycles. The period can be computed in the same way, considering the latencies count over a cycle instead of the token count. In figure 1(c), the graph is 3 periodic with a period 5 (as its critical size is the rightmost loop with a total of 5 latencies and 3 tokens).

In contrast to these results on the stationary periodic phase, little is known on the length of the initialization phase that leads to it.

2.4 Explicit schedules

It will be important in the sequel to use an explicit syntactic notation for handling schedules as modeling objects. We borrowed this notational idea from the theory of N-synchronous processes [7], where it is used to type programs with their schedule (that is, an expression representing the sequence of their firing instants). We use it in a slightly different way, to provide the schedules for each computation and transportation nodes in the Computation Net. As in their work, we focus on periodic schedules, as we are dealing with periodic behaviors.

Definition 2 (Schedules). A *schedule* for a computation net is a function $N \rightarrow w_N$ assigning an infinite word $w_N \in \{0,1\}^\omega$ to every computation and transportation node of the net. The intuition is that a schedule forces activity at instants where it holds a "1", and inactivity when "0". A schedule is said to be periodic when each such word w is of the form $w = u(v)^\omega$ where $u, v \in \{0,1\}^\star$. u is called the initial part, and v the periodic one. We call the length of v (noted $|v|$) the period of w, and the number of 1s in v, noted $|v|_1$, the periodicity of w.
Assuming that for all computation/transportation nodes, all w_N have identical period and periodicity, we note respectively p and k as the period and periodicity of the net. The *rate* r can then be defined as $\frac{k}{p}$.
A schedule is admissible if the w_N are mutually related in a way that respects the firing rule semantics of the model (from any given initial configuration of tokens there is only one admissible scheduling, following the deterministic synchronous semantics).

As an example, figure 1(d) shows the use of such schedule notation (we note $u(v)$ instead of $u.v^\omega$ to keep to ASCII notations). All local firing sequences are displayed, such as obtained by running our simple computation net. As was to be expected, the periodic parts are of length 5 (the period) and contain each 3 occurrences of symbol 1 (the periodicity). The explicit schedule allows in addition to these figures to visualize also the distribution of firings inside the period at each node. Ideally, if the system was well-balanced on latencies, the schedule of a given computation nodes should exactly be the one of its predecessor shifted right by one position (in a modulo fashion). But when data do *not* arrive synchronously at a given computation nodes and some have to be stalled at the entry, it is not the case. In our example this can occur only at the topmost computation nodes. Here we prefixed some of the inactivity 0 marks by a symbol to indicate that inaction is due to a lack of input token from the right ('), or on the left (‘).

3 Equalization Process

We now seek a very specific type of static periodic scheduling, obtained by inserting extra additional latencies on travel sections that are faster than others. The goal is to try and make the various travel paths as even as possible in terms of latency durations, so that data arrive simultaneously at their common destination. This should *not* further penalize the global speed of the net, which originally operates at the rate of its slowest loop cycle.

The extra latencies are virtual, in the sense that they do not correspond to physical lay-out demands, as the former are supposed to. In a methodological framework they could be used to redesign some of the components, for instance absorbing them as part of a more relaxed versions of computing elements.

Unfortunately, as our example of figure 1(b) shows, exact latency equalization is not always possible, at least with integer latency values. The leftmost loop is of rate $2/3$, and the (slower) rightmost one is of rate $3/5$. But adding one extra integer latency to the left loop brings its rate down to $\frac{2}{3+1} = 1/2$, strictly slower than the former slowest one.

Our approach will consist in inserting first as many integer delays as feasible, and then only add specific *fractional register (FR)* synchronous elements at appropriate places to correct the few equalization mismatches. FR elements should hold back temporarily "some" tokens (but not each one each turn). Our belief is that the preliminary integer completion, together with the smoothing effect of the initialization phase, are instrumental in spreading the token distribution even. If so, FR elements are reduced to simple, partially transparent latches (see next section on correctness issues).

In our running example of figure 1 (d) an intermediate FR element would be needed at the grey location on the leftmost loop, to hold back a token each time the topmost computation node indicates a $0'$ schedule activity, meaning at in this very instant the rightmost channel is not ready to deliver its value yet.

Next, we shall describe the successive algorithmic steps involved in the process of equalization. These steps are: *determining the expected global throughput*; *computing a maximal set of integer latencies* (to add corresponding elements to the system description; *computing the transitory and stationary schedules* (by linear state space construction); *adding the fractional latency elements*. These steps rely most of the time on well-known graph-theoretical and operational research techniques, but they sometimes have to be adapted to our goals. In the future other (optional) algorithmic techniques can be studied, mostly to shorten the initialization sequence of computation and transportation steps, making it asynchronous.

We describe the successive steps:

Global throughput evaluation. We need to compute the best feasible global rate, which is the slowest rate (noted R) amongst individual loop cycles. For this we do enumerate all elementary cycles, this is known as the *Minimal Cost-to-Time Ratio Cycle problem* [11, 9].

Integer latency insertion. This is solved by linear programming techniques. Linear equation systems are built to express that all elementary cycles, with possible extra variable latencies on arcs, should now be of rate R, the previously computed global throughput. The equations are also formed while enumerating the cycles in the previous phase. An additional requirement entered to the solver can be that the sum of added latencies be minimal (so they are inserted in a best factored fashion).

Rather than computing a rational solution and then extracting an integer approximate value for latencies, the particular shape of the equation system lends itself well to a direct *greedy* algorithm, stuffing incremental additional integer latencies into the existing systems until completion. This was confirmed by our prototype implementations.

The following example of figure 2 shows that our integer completion does *not* guarantee that all elementary cycles achieve a rate very close to the extremal, as explained in the caption. But this is here because a cycle "touches" the slowest one in several distinct locations.

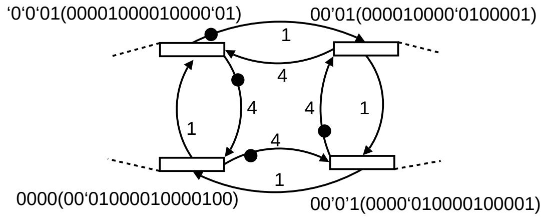

Figure 2. An example where no integer latency insertion can bring all the cycle rates very close to the global throughput. While the global throughput is of $\frac{3}{16}$, given by the inner cycle, no integer latency can be added to the outside cycle to bring its rate to $\frac{1}{5}$ from $\frac{1}{4}$. Instead four fractional latencies should be added (in each arc of weight 1).

Schedule computation (using state space construction). In order to compute the explicit schedules of the initial and stationary phases we currently need to *simulate* the system's behavior. We also need to store visited state, as a termination criterion for the simulation whenever an already visited state is reached. The purpose is to build (simultaneously or in a second phase) the schedule patterns of computation nodes, including the quote marks (’) and (‘), so as to determine where residual fractional latency elements have to be inserted.
In a synchronous run each state will have only one successor, and this process stops as soon as a state already encountered is reached back. The main issue here consists in the state space representation (and its complexity). In a naive approach each place may hold T tokens, where T is the minimal sum of tokens over all elementary cycles that use this place. But with the extra latencies now added, we can use LID modeling, with relay-stations and back-pressure mechanisms. Then each place can hold at most two tokens, encoded by two boolean values. Then a global state is encoded as $2n$ boolean variables, where n is the sum all all latencies over the arcs. Further simplification of the state

space in symbolic BDD model-checking fashion is also possible due to the fact that, in between the two values describing a relay-station state, one can be filled only if the other one already is. Internal implementation details are out of the scope of this paper.

We are currently investigated (as "future work") analytic techniques so as to estimate these phases without relying on this state space construction.

Fractional latencies. In an ideally equalized system, the schedules of distinct computation/transportation nodes should be precisely related: the schedule of the "next" node should be that of the "previous" node shifted one slot right (and the first schedule value of the target node is irrelevant here, as it is initial and not computed form the previous node). After we compute the effective schedules, one can whether this is the case. If not, then extra fractional registers need to be inserted just after the regular register already set between the nodes. This FR element should delay discriminatorily some tokens (but not all).

We shall introduce a formal model of our FR elements in the next subsection. The block diagram of its interfaces are displayed in figure 3.

We conjecture that, after integer latency equalization, such elements are only required just before computation nodes to where cycles with different original rates reconverge. We prove in subsection 3.2 that this is true under general hypothesis on smooth distribution of tokens along critical cycles. In our prototypal approach we have decided to allow them wherever the previous step indicated the need.

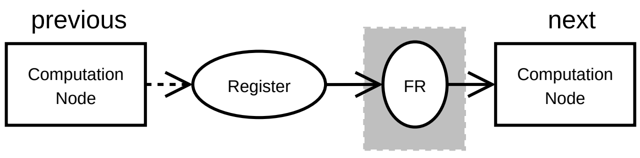

Figure 3. Fractional register insertion in the Network.

Optimized initialization. So far we have only considered the case where all components did fire as soon as they could. Sometimes delaying some computations or transportations in the initial phase could lead faster to the stationary phase, or even to a distinct stationary phase that may behave more smoothly as to its scheduling. Consider in the example of figure 1 (c) the possibility of firing the lower-right transport node alone (the one on the backward up arc) in a first step. By this one reaches immediately the stationary phase (in its last stage of iteration).

Initialization phases may require a lot of buffering resources temporarily, that will not be used anymore in the stationary phase. Providing short and buffer-efficient initialization sequences is thus a challenge. We are currently experimenting with symbolic *asynchronous* simulation (something akin to model-checking), trying to reach a given state known to be a stationary source as soon as possible. The asynchronous firing rule allows to perform various computations independently, so that the tokens may progress to better locations. Then a careful study of various paths may help choose the ones that use the less buffering resources. Other perspectives are open for future work here.

When applying these successive transformation and analysis steps, which may look quite complex, it is predictable that simple sub-cases often arise, due to the well-chosen numbers provided by the designer. Exact integer equalization is such a case. The case when fractional adjustments only occur at reconvergence to a critical paths are also noticeable. We built a prototype implementation of the approach, which indicates that these specific cases are indeed often met in practice.

3.1 Fractional Register element (FR)

We now formally describe the specific FR synchronous elements, both as a synchronous circuit in figure 4(b) and as a corresponding Mealy FSM in figure 4(a).

The FR interface consists of two input wires *TokenIn* and *Hold*, and one output wire *TokenOut*. Its internal state consists of a register *CatchReg*. The register will be used to "kidnap" the token (and its value in a real setting) for one clock cycle whenever *Hold* holds. We note $pre(CatchReg)$ the (boolean) value of the register computed at the previous clock cycle. It indicates whether the slot is currently occupied or free.

It is possible that the same token is held several instants in a row. But meanwhile there should be no new token arriving, as the FR element can store only one value; otherwise this would cause a conflict.
It is also possible that a full sequence of consecutive tokens are held back one instant each in a burst fashion. But then each token/value should leave the element in the very next instant to be consumed by the subsequent computation node; otherwise this would also cause a conflict.
Stated formally, when $Hold \wedge pre(CatchReg)$ holds then either $TokenIn$ holds, in which case the new data token enters and the current one leaves (by scheduling consistency the computation nodes that consumes it should then be active), or $TokenIn$ does not hold, in which case the current token remains (and, again by scheduling consistency, then the computing node should be inactive). Furthermore the two extra conditions are requested:

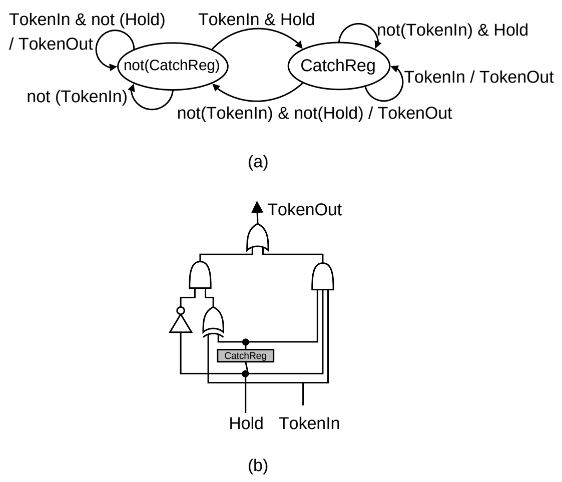

Figure 4. The automaton (a) and the interface block-diagram of the FR element

[$Hold \Rightarrow (TokenIn \vee pre(CatchReg))$:] if nothing can be held, the scheduling does not attempt to;

[$(TokenIn \wedge pre(CatchReg)) \Rightarrow Hold$:] otherwise the two tokens could cross the element and be output simultaneously.

The FR behavior amounts to the two equations:

[$CatchReg = Hold$:] the register slot is used only when the scheduling demands;

[$TokenOut = TokenOut_1 \vee TokenOut_2$:]

- $TokenOut_1 = TokenIn \oplus pre(CatchReg) \wedge \neg Hold$.
- $TokenOut_2 = TokenIn \wedge pre(CatchReg) \wedge Hold$.

either a new value directly falls across, or an old one is chased by a new one being held in its place.

Our main design problem is now to generate $Hold$ signals exactly when needed to respect the previous constraints. In addition it should be generated from the schedules of the source and target computation or transport nodes, to bridge from the former to the latter.

Consider again figure 3, we shall name w the schedule of the *previous* source node, and w' the schedule of the *next* target node. After the regular register delay the tokens are produce to the FR entry on schedule $0.w$ (shifted one slot/instant right). The fractional buffer should hold the token exactly when the k^{th} active step at this entry is *not* the k^{th} activity step at its target node that must consume it. In other words the FR element resynchronize its input and output, which cannot be away be more than one activity step. This last property is true as the schedules were computed using the LID approach with relay-stations, which do not allow more than one extra token in addition to the regular one on each arc between computation or transportation nodes.

Stated formally, this property becomes: $HOLD(n) = 1\ IFF\ |0.w_n|_1 \neq (|w'_n|_1 - |w'_0|_1)$. It says that at a given instant n we should kidnap a value if the number of occurrences of 1 up to instant n on the previous node is different than the number of occurrences of 1 on the next computation node. More precisely, the $-|w'_0|_1$ term takes care of a possible initial activity at the target node, not caused by the propagation of tokens from the source node, that would have to be removed.

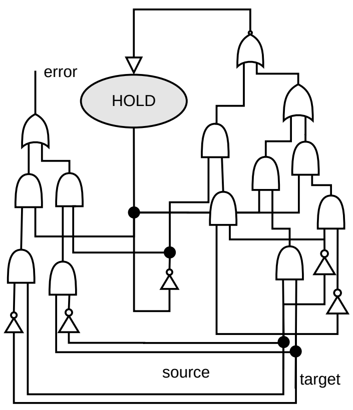

Figure 5. $Hold$ implementation.

Figure 5 shows a possible implementation computing $Hold$ from signals that would explicit provide the target and source schedules as inputs. It also computes an $Error$ output, which allows us to prove by various model-checking means that this particular signal can never be emitted.

3.2 Correctness issues

As already mentioned we still do not have a proof that in the stationary phase it is enough to include such elements at the entry points of computation nodes only, so they can be installed in place of more relay-stations also. Furthermore it is easy to find initialization phases where tokens in excess will accumulate at any locations, before the rate of the slowest loop cycles distribute them in a smoother, evenly distributed pattern. Still we have several hints that partially deal with the issue. It should be remembered here that, even without the result, we can equalize latencies (it just needs adding more FR elements).

Definition 3 (Smoothness). A schedule is called smooth if the sequences of successive 0 (inactive) instants in the schedule in between two consecutive 1 cannot differ by more than 1. The schedule $(1001)^\star$ is *not* smooth since they are two consecutive 0 between the first and second occurrences of 1, while there is none between the second and the third.

Property 1. *If all computation node schedules are smooth, rates can be equalized using FR elements only at computation node entry points.*

Proof. Suppose a vertex have token on standby in FR of input(s), but some token on other input(s) are absent(s). So, the maximal waiting time of absent token (n) is inferior or equal to the minimal distance (in clock cycle) between a present token and its following (n). In the worst case, when the following arrive, all previous are present and fire the transition. The condition of correctness on FR is preserved, a storage capacity of 1 is enough. □

4 Further Topics

We are currently implementing a prototype version. While some of the graph-theoretic algorithms are well-documented in the literature, the phase of symbolic simulation and state-space traversal needs careful attention to the representation of states (as Binary Decision Diagrams on variables encoding the local Relay-station states). Also the representation of schedules must be tuned. They are really strictly needed only at computation nodes, and can be recovered on demand at other places (the transportation nodes).

A number of important topics are left open for further theoretical developments:

- We certainly would like to establish that FR elements are needed only at computation nodes, minimizing their number rather intuitively;

- Discovering short and efficient initial phases is also an important issue here;

- The distribution of integer latencies over the arcs could attempt to minimize (on average) the number of computation nodes that are active altogether. In other words transportation latencies should be balanced so that computations alternate in time whenever possible. The goal is here to avoid *"hot spots"*. It could be achieved by some sort of retiming/recycling techniques;

- While we deal with *transportation* latencies, in general there can also be *computation* latencies. It can be encoded in our model with $\{$*begin/end*$\}$ refined operations, but one could introduce "less constant" computation latencies and pipeline stages;

- Marked graphs do not allow for control-flow alternatives and control *modes*. One reason is that, in a generalized setting such as full Petri Nets, it can no longer be asserted that token are consumed and produced at the same rate. But explicit *"branch schedules"* could maybe help regulate the branching control parts similarly to the way they control the flow rate.

Last but not least, we should soon conduct real case studies to validate the approach on industrial examples .

References

[1] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[2] Benveniste, Caspi, Edwards, Hallbwachs, L. Guernic, and de Simone. The synchronous languages twelve years later. IEEE and INRIA/IRISA, 2003.

[3] J. Boucaron, J.-V. Millo, and R. de Simone. Another glance at relay stations in latency-insensitive designs. In *FMGALS'05*, 2005.

[4] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and L. . D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verifi cation (CAV)*, page 12. UC Berkeley, Cadence Design Laboratories, July 1999.

[5] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *DAC'2004*, 2004.

[6] J. C. P. Chrétienne. *Problème d'ordonnancement: modélisation, complexité, algorithmes*. Masson, Paris, 1988.

[7] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks. In *POPL 2006 Proceedings*, January 2006.

[8] F. Commoner, A. W.Holt, S. Even, and A. Pnueli. Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, october 1971.

[9] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[10] V. H. V. Dongen, G. R. Gao, and Q. Ning. A polynomial time method for optimal software pipelining. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing*, volume 634 of *LNCS*, pages p613–624, 1992.

[11] E. Lawler. *Combinatorial Optimization: Network and Matroids*. Holt, Rinehart and Winston, 1976.

[12] L. P.Carloni, K. L.McMillan, and A. L.Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[13] L. P.Carloni, K. L.McMillan, A. Saldanha, and A. L.Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *THE BEST OF ICAD*, 200x.

[14] L. P.Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in soc design. *IEEE Micro*, 22(5):24–35, September/October 2002.

[15] F. R.Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer. Optimal design of synchronous circuits using software pipelining. In *Proceedings of the ICCD'98*, 1998.