Formal Methods for Scheduling of Latency-Insensitive Designs [*]

Julien Boucaron
AOSTE Project
INRIA Sophia-Antipolis

Robert de Simone
AOSTE Project
INRIA Sophia-Antipolis

Jean-Vivien Millo
AOSTE Project
INRIA Sophia-Antipolis

August 27, 2007

Abstract

LID (*Latency-Insensitive Design*) theory was invented to deal with SoC *timing closure* issues, by allowing arbitrary fixed integer latencies on long global wires. Latencies are coped with using a *resynchronization* protocol that performs dynamic scheduling of data transportation. Functional behaviour is preserved.

This dynamic scheduling is implemented using specific synchronous hardware elements: *Relay-Stations* (*RS*) and *Shell-Wrappers* (*SW*). Our first goal is to provide a formal modeling of *RS* and *SW*, that can then be formally verified.

As turns out, resulting behaviour is k-periodic, thus amenable to *static* scheduling. Our second goal is to provide formal hardware modeling here also. It initially performs *Throughput Equalization*, adding integer latencies wherever possible; residual cases require introduction of *Fractional Registers* (*FRs*) at specific locations.

Benchmark results are presented, run on our *KPassa* tool implementation.

1 Introduction

Long wire interconnect latencies induce time-closure difficulties in modern SoC designs, with propagation of signals across the die in a single clock cycle is problematic. The theory of *Latency-Insensitive Design (LID)*, proposed originally by Luca Carloni, Kenneth McMillan and Alberto Sangiovanni-Vincentelli [7, 8], offers solutions for this issue. This theory can roughly be described as such: an initial fully synchronous reference specification is first desynchronized as an asynchronous network of synchronous block components (a *GALS* system); it is then re-synchronized, but this time with proper interconnect mechanisms allowing specified (integer-time) latencies.

Interconnects consist of fixed-sized lines of so-called *Relay-Stations*. These *Relay-Stations*, together with *Shell-Wrapper* around the synchronous *Pearl* IP blocks, are in charge of managing the signal value flows. With their help proper regulation of the signal traffic is performed. Computation blocks may be temporarily paused at times, either because of input signal unavailability, or because of the inability of the rest of the network to store their outputs if they were produced. This latter issue stems from the limitation of fixed-size buffering capacity of the interconnects (*Relay-Station* lines).

Since their invention *Relay-Stations* have been a subject of attention for a number of research groups. Extensive modeling, characterization and analysis were provided in [9, 14, 13].

We mentioned before that the process of introducing latencies into synchronous networks introduced, at least conceptually, an intermediate asynchronous representation. This corresponds to *Marked Graphs* [16], a well-studied model of computation in the literature. The main property of *Marked Graph* is the absence of choice which matches with the absence of control in *LID*.

Marked Graphs with latencies were also considered under the name of *Weighted Marked Graphs (WMG)*[19]. We shall reduce *WMGs* to ordinary *Marked Graphs* by introducing new intermediate *Transportation Nodes (TN)*, akin to the previous *Computation Nodes (CN)* but with a single input and output *link*. In fact *LID* systems can be thought of as *WMGs* with buffers of capacity 2 (exactly) on *link* between *Computation and/or Transportation Nodes*. The *Relay-Stations* and *Shell-Wrappers* are an operational means to implement the corresponding flow-control and congestion avoidance mechanisms with explicit synchronous mechanisms.

The general theory of *WMG* provides many useful insights. In particular it teaches us that there exists static repetitive scheduling for such computational

*Work partially supported by ST Microelectronics and Texas Instruments grants in the context of the French regional PACA CIM initiative

behaviors [6, 2]. Such static k-periodic schedulings have been applied to software pipelining problems [18, 5], and later SoC *LID* design problems in [12]. 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. We shall try to provide a solution that "perfectly" equalizes latencies over reconvergent paths, so that token always arrive simultaneously at the *Computation Node*. Sadly, this cannot always be done by inserting an integer number of latency under the form of additional transportation sections. One sometimes need to hold back token for one step discriminatingly ans sometimes does not. We provide our solution here under the form of *Fractional Registers (FR)*, that may hold back values according to an (input) regular pattern that fits the need for flow-control. Again we contribute explicit synchronous descriptions of such elements, with correctness properties. We also rely deeply on a syntax for schedule representation, borrowed from the theory of *N-synchronous processes* [15].

Explicit static scheduling that uses predictable synchronous elements is desirable for a number of issues. It allows *a posteriori* precise re-dimensioning of glue buffering mechanisms between local synchronous elements to allow the system to work, and this without affecting the components themselves. Finally, the extra virtual latencies introduced by equalization could be absorbed by the local computation times of *CN*, to resynthesize them under relaxed timing constraints.

We built a prototype tool for equalization of latencies and *Fractional Registers* insertion. It uses a number of elaborated graph-theoretical and linear-programming algorithms. We shall briefly describe this implementation.

Contributions: Our first contribution is to provide a formal description of *Relay-Stations* and *Shell-Wrappers* as synchronous elements [4], something that was never done before in our knowledge (the closest effort being [10]). We introduce local correctness properties that can easily be model-checked; these generic local properties, when combined, ensure the global property of the network.

We introduce the *Equalization process* to statically schedule a *LID* Specification: slowing down *"too fast"* cycles while maintaining the original throughput of the *LID* Specification. **The goal is to simplify the *LID* protocol**.

But rational difference of rates may still occur after *Equalization process*, we solve it by adding *Fractional Registers (FR)*, that may hold back values according to a regular pattern that fits the need for flow-control.

We introduce a new class of *smooth* schedules that optimally-minimizes the number of *FRs* used on a statically scheduled *LID* design.

Article Outline: In the next Section we provide some definitional and notational background on various models of computations involved in our *modeling framework*, together with an explicit representation of periodic schedules and firing instants; with this we can state historical results on k-periodic scheduling of *WMGs*. In Section 3 we provide the synchronous reactive representation of *Relay-Stations* and *Shell-Wrappers*, show their use in dynamic scheduling of *Latency-Insensitive Design*, and describe several formal local correctness properties that help with the global correctness property of the full network. Statically scheduled *LID* systems are tackled in Section 4; we describe an algorithm to build a statically scheduled *LID*, possibly adding extra virtual integer latencies and even *Fractional Registers*. We provide a running example to highlight potential difficulties. We also present benchmarks result of a prototype tool which implements the previous algorithms and their variations. We conclude with considerations on potential further topics.

2 Modeling Framework

2.1 Computation nets

We start from a very general definition, describing what is common of all our models.

Definition 1 (Computation Network Scheme). We call *Computation Network Scheme (CNS)* a graph whose vertices are called *Computation Nodes*, and whose arcs are called *links*. We also allow arcs without a source vertex, called *input links*, or without target vertex, called *output links*.

An instance of a *CNS* is depicted on Figure 1 (a).

The intention is that *Computation Nodes* perform computations by **consuming a data on each of its incoming *links*, and producing as a result a new data on each of its outgoing *links***.

The occurrence of a computation thus only depends on data presence and not their actual values, so that data can be safely abstracted as *tokens*. A *CNS* is choice free.

In the sequel we shall often consider the special case where the *CNS* forms a strongly connected graph, unless specified explicitly.

This simple model leaves out the most important features, that are mandatory to define its operational semantics under the form of behavioral firing rules. Such features are:

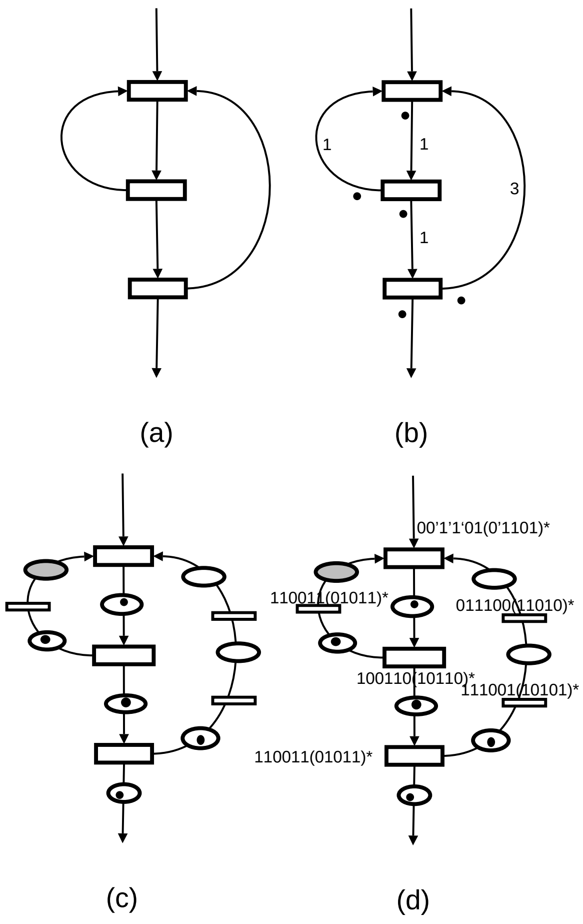

Figure 1: (a) An example of *CNS* (with rectangular *Computation Nodes*), (b) a corresponding *WMG* with latency features and token information, (c) a *SMG/LID* with explicit (rectangular) *Transportation Nodes* and (oval) *places*/*Relay-Stations*, dividing arcs according to latencies, (d) a *LID* with explicit schedules

- the *initialization* setting (where do tokens reside initially),
- the nature of *links* (combinatorial wires, simple registers, bounded or unbounded *place*, ...),
- and the nature of *time* (synchronous, with computations firing simultaneously as soon as they can, or asynchronous, with distinct computations firing independently).

Setting up choices in these features provides distinct Models of Computation.

2.2 Synchronous/asynchronous versions

Definition 2. A *Synchronous Reactive Net* (*S/R* net) is a *CNS* where time is synchronous: all *Computation Nodes* fire simultaneously. In addition *links* are either (memoryless) combinatorial wires or simple registers, and all such registers initially hold a token.

The *S/R* model conforms to synchronous digital circuits or (single-clock) synchronous reactive formalisms [3]. The network operates "at full speed": there is always a value present in each register, so that *CNs* operates at each instant. As a result, they consume all values (from registers and through wires), and replace them again with new values produced in each register. The system is *causal* iff there is at least one register along each cycle in the graph. Causal *S/R* nets are well-behaved in the sense that their semantics is well-founded.

Definition 3. A *Marked Graph* is a *CNS* where time is asynchronous: computations are performed independently, provided they find enough tokens in their incoming *links*; *links* have a *place* holding a number of *tokens*; in other words, *Marked Graphs* form a subclass of Petri Nets. The initial marking of the graph is the number of tokens held in each *place*. In addition a *Marked Graph* is said to be *of capacity k* if each *place* can hold no more than k tokens.

There is a simple way to encode *Marked Graphs* with capacity as *Marked Graphs* with unbounded capacity: this requires to add a reverse *link* for each existing one, which contains initially a number of tokens equal to the difference between the capacity and the initial marking of the original *link*.

It was proved that a strongly connected *Marked Graph* is live (each computation can always be fired in the future) iff there is at least one token in every cycle in the graph [16]. Also, the total number of tokens in a cycle is an invariant, so strongly connected *Marked Graphs* are k-safe for a given capacity k.

Under proper initial conditions *S/R* nets and *Marked Graphs* behave essentially the same, with *S/R* systems performing all computations simultaneously "at full rate", while similar computations are now performed independently in time in *Marked Graph*.

Definition 4. A *Synchronous Marked Graph (SMG)* is a *Marked Graph* with an *ASAP (As Soon As Possible)* semantics: each *Computation Node* (transition) that may fire due to the availability of it input tokens immediately does so (for the current instant).

SMGs and the *ASAP* firing rule are underlying the works of [6, 2], even though they are not explicitly given name there.

Figure 1 (c) shows a *Synchronous Marked Graph*. Note that *SMGs* depart from *S/R* models: here all tokens are not always available.

2.3 Adding latencies and time durations

We now add latency information to indicate transportation or computation durations. These latencies shall be all along constant integers (provided from "outside").

Definition 5. A *Weighted Marked Graph (WMG)* is a *CNS* with (constant integer) latency labels on *links*. This number indicates the time spent while performing the corresponding token transportation along the *link*.

We avoid computation latencies on *CNs*, which can be encoded as transportation latencies on *links* by splitting the actual *CN* into a `begin/end_CN`. Since latencies are global time durations, the relevant semantics which take same into account is necessarily *ASAP*. The system dynamics also imposes that one should record at any instant "how far" each token is currently in its travel. This can be modeled by an age stamp on token, or by expanding the *WMG* links with new *Transportation Nodes (TN)* to divide them into as many sections of unit latency. *TNs* are akin to *CNs*, with the particularity that they have unique source and target links. This expansion amounts to reducing *WMGs* to (much larger) plain *SMGs*. Depending on the concern, the compact or the expanded form may be preferred.

Figure 1 (b) displays a *Weighted Marked Graph* obtained by adding latencies to figure (a), which can be expanded into the *SMG* of figure (c).

For correctness matters there should still be at least one token along each cycle in the graph, and less token on a *link* than its prescribed latency. This corresponds to the correctness required on the expanded *SMG* form.

Definition 6. A *Latency-Insensitive Design (LID)* is a *WMG* where the expanded *SMG* obtained as above uses *places* of capacity 2 in between *CNs* and *TNs*.

This definition reads much differently than the original one in [8]. This comes partly from an important concern of the authors then, which is to provide a description built with basic components (named *Relay-Stations* and *Shell-Wrappers*) that can easily be implemented in hardware. Next Section 3 provides a formal representation of *Relay-Stations* and *Shell-Wrappers*, together with their properties.

Summary *CNS* lead themselves quite naturally to both synchronous and asynchronous interpretations. Under some easily expected initial conditions, these variants can be shown to provide the same input/output behaviours. With explicit latencies to be considered in computation and data transportation this remains true, even if congestion mechanisms may be needed in case of bounded resources. The equivalence in the ordering of event between a synchronous circuit and a *LID* circuit is shown in [7], and equivalence between a *MG* and a *S/R* design is shown in [20].

2.4 Periodic behaviors, throughput and explicit schedules

We now provide the definitions and classical results needed to justify the existence of static scheduling. This will be used mostly in Section 4, when we develop our formal modeling for such scheduling using again synchronous hardware elements.

Definition 7 (Rate, throughput and critical cycles). Let G be a *WMG* graph, and C a cycle in this graph.
The *rate* R of the cycle C is equal to $\frac{T}{L}$, where T is the number of tokens in the cycle, and L is the sum of latencies of the arcs of this given cycle.
The *throughput* of the graph is defined as the minimum rate among all cycles of the graph.
A cycle is called *critical* if its rate is equal to the *throughput* of the graph.

A classical result states that, provided simple structural correctness conditions, a strongly-connected *WMG* runs under a ultimately k-periodic schedule, with the throughput of the graph [6, 2]. We borrow notation from the theory of N-*synchronous processes* [15] to represent these notions formally, as explicit analysis and design objects.

Definition 8 (Schedules, periodic words, k-periodic schedules). A *pre-schedule* for a *CNS* is a function $Sched : N \rightarrow w_N$ assigning an infinite binary word $w_N \in \{0,1\}^\omega$ to every *Computation Node* and *Transportation Node* N of the graph. *Node* N is *activated* (or triggered, or fired, or run) at global instant i iff $w_N(i) = 1$, where $w(i)$ is the i^{th} letter of word w.

A pre-schedule is a *schedule* if the allocated activity instants are in accordance with the token distribution (the lengthy but straightforward definition is left to the reader). Furthermore, the schedule is called *ASAP* if it activates a node N whenever all its input tokens have arrived (according to the global timing).

An infinite binary word $w \in \{0,1\}^\omega$ is called *ultimately periodic*: if it is of the form $u.(v)^\omega$ where u and $v \in \{0,1\}^\star$, u represents the initialization phase, and v the periodic one.
The *length* of v is noted $|v|$ and called its *period*. The number of occurrences of 1s in v is denoted $|v|_1$ and called its *periodicity*. The *rate* R of an ultimately periodic word w is defined as $\frac{|v|_1}{|v|}$.

A schedule is called k-periodic whenever for all N, w_N is a periodic word.

Thus a schedule is constructed by simulating the *CNS* according to its (deterministic) *ASAP* firing rule.

Furthermore, it has been shown in [2] that the length of the stationary periodic phase (called period) can be computed based on the structure of the graph and the (static) latencies of cycles: for a $CSCC$ (Critical Strongly Connected Component) the length of the stationary periodic phase is the GCD (Greatest Common Divisor) over latencies of its critical cycles. For instance assume a $CSCC$ with 3 critical cycles having the following rates: $2/4, 4/8, 6/12$, the GCD of latencies over its critical cycles is: 4. For the graph, the length of its stationary periodic phase is the LCM (Least Common Multiple) over the ones computed for each $CSCCs$. For instance assume the previous $CSCC$ and another one having only one critical cycle of *rate* $1/2$ then the length of the stationary periodic phase of the whole graph is 2.

Figure 1(d) shows the schedules obtained on our example. If latencies were *"well-balanced"* in the graph, tokens would arrive simultaneously at their consuming node; then, the schedule of any *Node* should exactly be the one of its predecessor(s) shifted right by one position. However it is not the case in general when some input tokens have to stall awaiting others. The "difference" (target schedule minus 1-shifted source schedule) has to be coped with by introducing specific buffering elements. This should be limited to the locations where it is truly needed. Computing the static scheduling this allows to avoid adding the second register that was formerly needed everywhere in *RSs*, together with some of the backpressure scheme.

The issue arises in our running example only at the top-most *Computation Node*. We indicate it by prefixing some of the inactive steps (0) in its schedule by symbols: lack of input from the right input *link* ('), or from the left one (').

3 Synchronous to LID: Dynamic Schedule

In this Section we shall briefly recall the theory of *Latency-Insensitive Design*, and then focus on formal modeling with synchronous components of its main features [4].

LID theory was introduced in [7]. It relies on the fact that *links* with latency, seen as physical long wires in synchronous circuits, can be segmented into sections. Specific elements are then introduced in between sections. Such elements are called *Relay-Stations (RS)*. They are instantiated at the oval places in Figure 1(c). Instantaneous communication is possible inside a given section, but the values have to be buffered inside the *RS* before it can be propagated to the next section. The problem of computing realistic latencies from physical wire lengths was tackled in [11], where a physical synthesis floor-planner provides these figures.

Relay-Stations are complemented with so-called *Shell-Wrappers (SW)*, which compute the firing condition for their local synchronous component (called *Pearl* in *LID* theory). They do so from the knowledge of availability of input token and output storage slots.

3.1 Relay-Stations

The signaling interface of a *Relay-Station* is depicted in Figure 2. The `val` signals are used to propagate tokens, the `stop` signal are used for congestion control. For symmetry here `stop_out` is an input and `stop_in` an output.

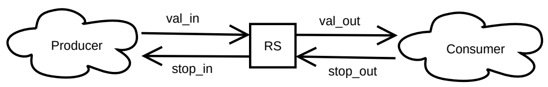

Figure 2: Relay-Station - Block Diagram

Intuitively the *Relay-Station* behaves as follows: when traffic is clear (no stop), each token is propagated down at the next instant from the one it was received. When a `stop_out` signal is received because of downward congestion, the *RS* keeps its token. But then, the previous section and the previous *RS* cannot be warned instantly of this congestion, and so the current *RS* can perfectly well receive another token at the same time it has to keep the former one. So there is a need for the *RS* to provide a second auxiliary register slot to store this second token. Fortunately there is no need for a third one: in the next instant the *RS* can propagate back a `stop_in` control information to preserve itself from receiving yet another value. Meanwhile the first token can be sent as soon as `stop_out` signals are withdrawn, and the *RS* remains with only one value, so that in the next step it can already allow a new one and not send its congestion control signal. Note that in this scheme there is no undue gap between the token sent.

This informal description is made formal with the description of a synchronous circuit with two registers describing the *RS* in Figure 3, and its corresponding syncchart [1] (in Mealy FSM style) in Figure 4.

The syncchart contains 4 states:

`empty` *when no token are currently buffered in the*

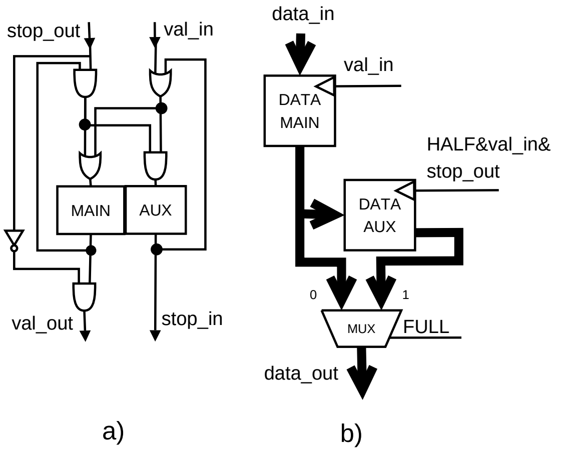

Figure 3: Relay-Station - a) Control Logic b) Data Path

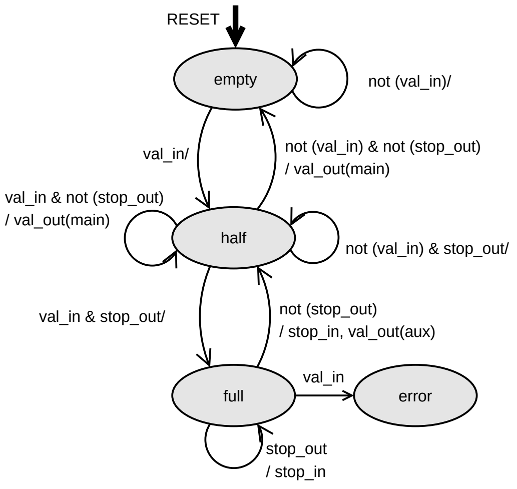

Figure 4: Relay-Station syncchart

RS; in this state the *RS* simply waits for a valid input token coming, and store it in its `main` register then it goes to state `half`. $stop\_out$ signals are ignored, and not propagated upstream, as this *RS* can absorb traffic.

`half` *when it holds one token*; Then the *RS* only transmits its current, previously received token if ever it does not receive an halting $stop\_out$ signal. If halting is requested, ($stop\_out$), then it retains its token, but must also accept a potential new one coming from upstream (as it has not sent any back-pressure holding signal yet). In the second case it becomes full, with the second value occupying its "emergency" `aux`iliary register. If the *RS* can transmit ($stop\_out = false$), it either goes back to `empty` or retrieve a new valid signal (val_in), remaining then in the same state. On the other hand it still makes no provision to propagate back-pressure (in the next clock cycle), as it is still unnecessary due to its own buffering capacity.

`full` *when it contains two tokens*; then it raises in any case the $stop\_in$ signal, propagating to the upstream section the hold-out $stop\_out$ signal received in the previous clock cycle. If it does not itself receive a new $stop\_out$, then the line downstream was cleared enough so that it can transmit its token; otherwise it keeps it and remains halted.

`error` is a state which should never be reached (in an *assume/guarantee* fashion). The idea is that there should be a general precondition stating that the environment will never send the val_in signal whenever the RS emits the $stop\_in$ signal. This should be extended to *any* combination of RS, and build up a "sequential care-set" condition on system inputs. The property is preserved as a postcondition as each RS will guarantee correspondingly that val_out is *not* sent when $stop\_out$ arrives.

NB: The notation $val\_out(main)$ or $val\_out(aux)$ means emit the signal val_out taking its value in the buffer, respectively, $main$ or aux.

Correctness properties Global correctness depends upon an assumption on the environment (see description of `error` state above). We now list a number of properties that should hold for *Relay-Stations*, and further *links* made of a connected line $L_n(k)$ of n successive *RS* elements and currently containing k values (remember that a line of n RS can store $2n$ values).

On a single *RS*:

- $\Box\ \neg(stop\_out \wedge val\_out)$ (back-pressure control takes action immediately);
- $\Box\ ((stop\_out \wedge X(stop\_out)) \Rightarrow X(stop\_in))$ (a stalled *RS* gets filled in two steps)

where $\Box$, $\Diamond$, $\mathcal{U}$ and X are the traditional *Always*, *Eventually*, *Until* and *Next* (linear) temporal logic operators. More interesting properties can be asserted on lines of *RS* elements (we assume that by renaming $stop\_\{in, out\}$ and $val\_\{in, out\}$ signals form the I/O interface of the global line $L_n(k)$):

- $\Box\ (\neg stop\_out \Rightarrow \neg X^n(stop\_in)\,)$ (free slots propagate backwards);
- $\Box\big((stop\_out \quad \mathcal{U} \quad X^{(2n-k)}(true)) \quad \Rightarrow X^{(2n-k)}(stop\_in)\big)$ (overflow);

- $\big(\Diamond\ val\_in \wedge \Box(\Diamond(\neg stop\_out)\big) \Rightarrow \Diamond val\_out)$ (if traffic is not completely blocked from below from a point on, then tokens get through)

The first property is true of any line of length n, the second of any line containing initially at least k tokens, the third of any line.

We have implemented *RSs* and lines of *RSs* in the `Esterel` synchronous language, and model-checked combinations of these properties using $EsterelStudio$[1] ™.

3.2 Shell-Wrappers

The purpose of *Shell-Wrappers* is to trigger the local *Computation Node* exactly when tokens are available from each *input link*, and there is storage available for result in *output links*. It corresponds to a notion of *clock gating* in circuits: the *SW* provides the logical clock that activates the IP component represented by the *CN*. Of course this requires that the component is physically able to run on such an irregular clock (a property called *patience* in *LID* vocabulary), but this technological aspect is transparent to our abstract modeling level. Also, it should be remembered that the *CN* is supposed to produce data on all its outputs while consuming on all its inputs in each computation step. This does not imply a combinatorial behavior, since the *CN* itself can contain internal registers of course. A more fancy framework allowing *computation latencies* in addition to our communication latencies would have to be encoded in our formalism. This can be done by "splitting" the node into a `begin_CN` and a `end_CN` nodes, and installing internal transportation links with desired latencies between them; if the outputs are produced with different latencies one should even split further the node description. We shall not go into further details here, and keep the same abstraction level as in *LID* and *WMG* theories.

The signal interface of *SWs* consists of `val_in` and `stop_in` signals indexed by the number of *input links* to the *SW*, and of `val_out` and `stop_out` signals indexed by the number of its *output links*. There is an output `clock` signal in addition, to fire the local component. This last signal will be scheduled at the rate of local firing thus. Note that it is here synchronous with all the `val_out` signals when values are abstracted into tokens.

The operational behavior of the *SW* is depicted as a synchronous circuit in Figure 5 (a), where each `Input i` module has to be instantiated with the Figure 5 (b), with its signals properly renamed, finally driving the data path in Figure 5 (c). The *SW* is combinatorial, it takes one clock cycle to pass from *RSs* before the *SW*, through the *SW* and its *Pearl*, and finish into *RSs* in outputs of the *SW*. The *Pearl* is *Patient*, the state of the *Pearl* is only changed when clock (periodic or sporadic) occurs.

[1] $EsterelStudio^{TM}$ is a trademark of $Esterel\ Technologies$

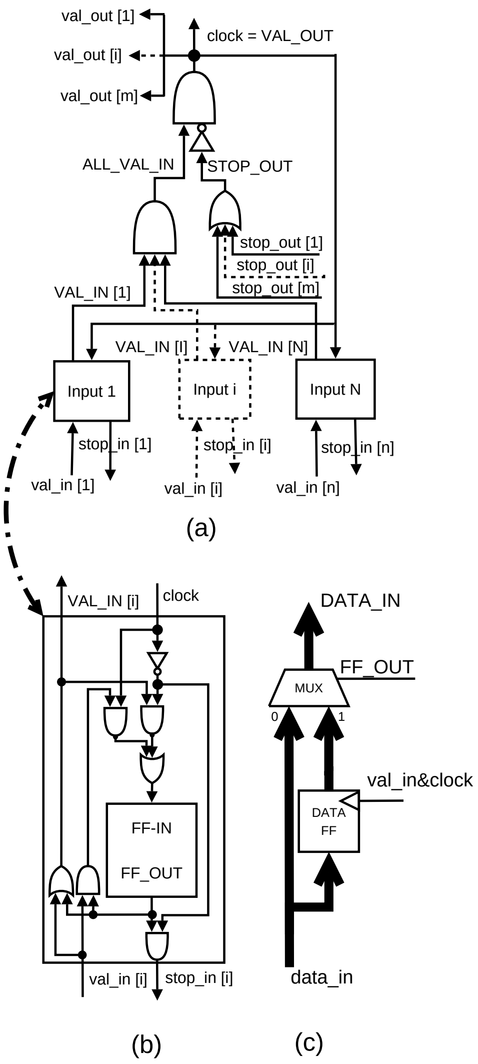

Figure 5: (a) Shell-Wrapper Circuitry, (b) Input module, and (c) Data Path

The *SW* works as follows:

- the internal *Pearl's clock* and all val_out_i valid output signals are generated once we have all val_in (signal ALL_VAL_IN in Figure 5 (a)), while $stop$ is false. The internal $stop$ signal itself represents the disjunction of all incoming $stop\_out_j$ signals from outcoming channels (signal $STOP\_OUT$ in Figure 5 (a));

- the buffering register of a given input channel is used meanwhile as long as not all other input tokens are available (Figure 5 (b));

- so, internal *Pearl's clock* is set to false whenever a backward $stop\_out_j$ occurs as true, or a forward val_in_i is false. In such case the registers already busy hold their *true* value, while others may receive a valid token “just now”;

- $stop\_in_i$ signals are raised towards all channels whose corresponding register was already loaded (a token was received before, and still not consumed), to warn them not to propagate any value in this clock cycle. Of course such signal cannot be sent in case the token is currently received, as it would raise a causality paradox (and a combinatorial cycle).

- flip-flop registers are reset when the *Pearl’s* clock is raised, as it consumes the input token. Following the previous remark, the signal $stop\_in_i$ holding back the traffic in channel i is raised for these channels where the token have arrived before the current instant, even in this case.

Correctness properties Again we conducted a number of model-checking experiments on *SWs* using ESTEREL STUDIO:

- $\Box\big((\exists j,\ stop\_out_j)\vee \Rightarrow \neg clock\big)$ where j is an input index;

- $\Box\big((\exists j,\ stop\_out_j) \Rightarrow (\forall i,\ \neg val\_out_i)\big)$ where j/i is an input/output index respectively;

- $\Box\big((\forall j,\ \neg stop\_out_j \wedge \neg X(stop\_out_j)) \Rightarrow (X(clock) \Rightarrow \exists i,\ X(val\_in_i))\big)$ where j, i are input index (if the SW was not suspended at some instant by output congestion, and it triggers its pearl the next instant, then it has to be because it received a new value toke on some input at this next instant)

On the other hand, most useful properties here would require syntactic sugar extensions to the logics to be easily formulated (like “a token has had to arrive on each input before or when the *SW* triggers its local *Pearl*”, but they can arrive in any order).

As in the case of RSs, correctness also depends on the environmental assumption that $\forall i,\ stop\_in_i \Rightarrow \neg val\_in_i$, meaning that upward components must not send a value while this part of the system is jammed.

3.3 Tool implementation

We built a prototype tool named KPASSA[2] to simulate and analyze a *LID* system made of a combination of previous components.

[2]stands for *K-Periodic Asap Schedule Simulation and Analysis*, pronounced “*Que pasa ?*”

Simulation is eased by the following fact: given that the *ASAP* synchronous semantics of *LID* ensures determinism, for closed systems each state has exactly one successor. So we store states that were already encountered to stop the simulation as soon as a state already visited is reached.

While we will come back to the main functions of the tool in the next Section, it can be used in this context of dynamic scheduling to detect where the back-pressure control mechanisms are really been used, and which *Relay-Stations* actually needed their secondary register slot to preserve from traffic congestion.

4 Synchronous to LID: Static Scheduling

We now turn to the issue of providing static periodic schedules for *LID* systems. According to the previous philosophy governing the design of *Relay-Stations*, we want to provide solutions where tokens are not allowed to accumulate into *places* in large numbers. In fact we will attempt to *equalize* the flows so that tokens arrive as much as possible simultaneously at their joint *Computation Nodes*.

We try to achieve our goal by adding new virtual latencies on some paths that are *faster* than others. If such an ideal scheme could lead to *perfect equalization* then the second buffering slot mechanism of *Relay-Stations* and the back-pressure control mechanisms could be done without altogether. However it will appear that this is not always feasible. Nevertheless integer latency equalization provides a close approximation, and one can hope that the additional correction can be implemented with smaller and simpler *Fractional Registers*.

Extra virtual latencies can often be included as computational latencies, thereby allowing the redesign of local *Computation Nodes* under less stringent timing budget.

As all connected graphs, general (connected) *CNS* consist of Directed Acyclic Graphs of strongly connected components. If there is at least one cycle in the net it can be shown that all cycles have to run at the rate of the slowest to avoid unbounded token accumulation. This is also true of input token consumption, and output token production rates. Before we deal with the (harder) case of strongly connected graphs that is our goal, we spend some time on the (simpler) case of acyclic graphs (with a single *input link)*.

4.1 DAG Case

We consider the problem of equalizing latencies in the case of Directed Acyclic Graphs (DAGs) with a single source *Computation Node* (one can reduce DAGs to this sub-case if all inputs are arriving at the same instant), and no initial token is present in the DAG.

Definition 9 (DAG Equalization). In this case the problem is to *equalize* the DAG such that all paths arriving to a *Computation Node* are having the *same latency* from inputs.

We provide a sketch of the abstract algorithm and its correction proof.

Definition 10 (Critical Arc). We define an arc as *critical* if it belongs to a path of maximal latency $Max_l(N)$ from the global source *Computation Node* to the target *Computation Node* N of this arc.

Definition 11 (Equalized Computation Node). We define a *Computation Node* N which is having only incoming *critical* arcs to be an *equalized Computation Node*, i.e from any path from the source to this *Computation Node* we have the same latency $Max_l(N)$.
If a *Computation Node* has only one incoming arc then this arc will be *critical* and this *Computation Node* will be *equalized* by definition.

The core idea of the algorithm is first to find for each *Computation Node* N of the graph what is its maximal latency $Max_l(N)$ and to mark incoming *critical* arcs; Then the second idea is to *saturate* all *non-critical* arcs of each *Computation Node* of the DAG in order to obtain an *equalized* DAG.

The first part of the algorithm is done through a modified *longest-path algorithm*, marking incoming *critical* arcs for each *Computation Node* of the DAG and putting for each *Computation Node* N its maximal latency $Max_l(N)$ (as shown in algorithm 1).

The second part of the algorithm is done as follows (see algorithm 2): Since it may exist incoming arcs of a *Computation Node* N that are not *critical*: it exists an ϵ integer number that we can add such that the non-*critical* arc becomes *critical*. We can compute this integer number ϵ easily through this formula: $Max_l(N) = Max_l(N') + non\_critical\_arc_l + \epsilon$, where N' is the source *Computation Node* passing through the *non-critical arc* and reaching the *Computation Node* N. Now, the non-*critical* arc through the add of ϵ is *critical*.
We apply this for all non-*critical* arcs of the *Computation Node* N, then the *Computation Node* is *equalized*.
Finally, we apply this for all *Computation Nodes* of the DAG, then the DAG is *equalized*.

Algorithm 1 procedure recursive_longest_path ($NODE$ source)

```
Require: Graph is a DAG
  for all $ARC\ arc\ of\ source.getOutputArcs()$ do
    $NODE$ node ⇐ arc.getTargetNode();
    $unsigned$ currentLatency ⇐ arc.getLatency() + source.getLatency();
    {if the latency of this path is greater}
    if $(node.getLatency() \leq currentLatency)$ then
      arc.setCritical($true$);
      node.setLatency(currentLatency);
      { update arcs critical field for "node" }
      for all $ARC\ node\_arc\ of\ node.getInputArcs()$ do
        if $(node\_arc.getLatency() + node\_arc.getSourceNode().getLatency() < currentLatency)$ then
          node_arc.setCritical($false$);
        else
          node_arc.setCritical($true$);
        end if
      end for
      {recursive call on "node" to update the whole sub-graph}
      recursive_longest_path(node);
    end if
  end for
```

Algorithm 2 procedure final_equalization ($GRAPH$ graph)

```
Require: Graph is a DAG
  for all $NODE\ node\ of\ graph.getNodes()$ do
    for all $ARC\ arc\ of\ node.getInputArcs()$ do
      if $(arc.isCritical() == false)$ then
        $unsigned$ maxL ⇐ node.getLatency();
        $unsigned$ $\epsilon$ ⇐ maxL - (arc.getLatency() + arc.getSourceNode().getLatency());
        arc.setLatency(arc.getLatency() + $\epsilon$);
        arc.setCritical($true$);
      end if
    end for
  end for
```

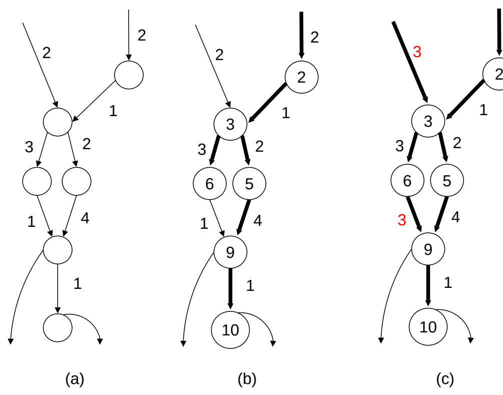

Figure 6: (a) An *unequalized*, (b) *critical* paths annotated (large *links*) and (c) *equalized* DAG

An instance of the *unequalized*, *critical* arcs annotated and *equalized* DAG is shown in Figure 6:

Starting from the *unequalized* graph in Figure 6 (a):

The first pass of the algorithm is determining for each *Computation Node*, its maximal latency Max_l (in circles) and incoming *critical* arcs denoted using **large** *links* as in Figure 6 (b).

The second part of the algorithm is adding *"virtual" latencies* (the ϵ) on non-*critical* incoming arcs, since we known what are the *critical* arcs coming through each *Computation Node* (**large** *links*), then we just have to add the needed amount (ϵ) in order that the non-*critical* arc is now *critical*: the sub between the value of the target *Computation Node*, minus the sum between the arriving *critical* arc and its source *Computation Node* maximal latency. For instance, consider the *Computation Node* holding a 9, the left branch is not *critical*, hence we are just solving $9 = 6 + 1 + \epsilon$ and $\epsilon = 2$ thus the arc will now have a latency of $3 = 1 + \epsilon$ and is so *critical* by definition. Finally the whole graph will be fully-*critical* and thus *equalized* by definition as in Figure 6 (c).

Definition 12. A *critical* path is composed only of *critical* arcs.

Theorem 1. *DAG equalization algorithm is correct*

Proof. For all *Computation Nodes*, there is at least one *critical* arc incoming by definition; then if there is more than one incoming arc, we add the result of the sub between the maximum latency of the path passing through the so-called *critical* arc and the add between the *non-critical* arc latency and the maximum latency of the path arriving to the *Computation Node* where the *non-critical* arc starts. Now any arc on this given *Computation Node* are all *critical* and thus this *Computation Nodes* is *equalized* by definition. And this is done for any *Computation Node*, thus the graph is *equalized*. Since in any case we do not modify any *critical* arc, we still have the same maximum latency on *critical* paths. □

4.2 Strongly Connected Case

In this case, the successive algorithmic steps involved in the process of **equalization** consist in:

1. Evaluate the graph *throughput*;
2. Insert as many additional integer latencies as possible (without changing the global throughput);
3. Compute the static schedule and its initial and periodic phases;
4. Place *Fractional Registers* where needed;
5. Optimize the initialization phase (optional)

These steps can be illustrated on our example in Figure 1.

1. The left cycle in Figure (b) has *rate* $2/2 = 1$, while the (slowest) rightmost one has *rate* $3/5$. *Throughput* is thus $3/5$;
2. A single extra integer latency can be added to the *link* going upward in the left cycle, bringing this cycle's rate to $2/3$. Adding a second one would bring the rate to $2/4 = 1/2$, slower than the global *throughput*. This leads to the expanded form in Figure 1 (c);
3. The *WMG* is still not equalized. The actual schedules of all *CN* can be computed (using KPASSA, as displayed in Figure (d). Inspecting closely those schedules one can notice that in all cases the schedule of a *CN* is the one of its predecessors shifted right by one position, **except** for the schedule of the topmost *Computation Node*. One can deduce from the differences in scheduling exactly when the additional buffering capacity was required, and insert dedicated *Fractional Registers* which delay selectively some tokens accordingly. This only happens for the initial phase for tokens arriving from the right, and periodically also for tokens arriving from the left;
4. It could be noticed that, by advancing only the single token at the bottom of the up going rightmost *link* for one step, one reaches immediately the periodic phase, thus saving the need for a *FR* element on the right cycle used only in the initial phase. Then only one *FR* has to be added past the regular latch register colored in grey.

We describe now the **equalization** algorithm steps in more details:

Graph throughput evaluation: For this we enumerate all elementary cycles and compute their *rates*. While this is worst-case exponential, it is often not the case in the kind of applications encountered. An alternative would be to use well-known "minimum mean cycle problem" algorithms (see [17] for a practical evaluation of those algorithms). But the point here is that we need all those elementary cycle for setting up Linear Programming (LP) constraints that will allow to use efficient LP solving techniques in the next step. We are currently investigating alternative implementations in KPASSA.

Integer latency insertion: This is solved by LP 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 7 shows that our integer completion does *not* guarantee that all elementary cycles achieve a rate very close to the extremal. But this is here because a cycle "touches" the slowest one in several distinct locations. 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).

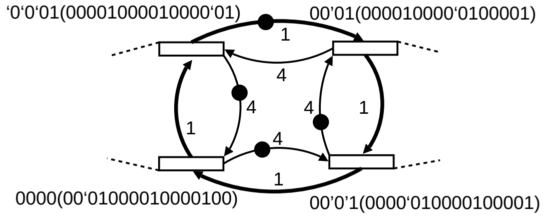

Figure 7: An example of WMG where no integer latency insertion can bring all the cycle rates the closest to the global throughput.

Initial and Periodic phase Schedule computation: 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). Further simplification of the state space in symbolic BDD model-checking fashion is also possible but it is out of the scope of this paper.

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

Fractional Register insertion: In an ideally equalized system, the schedules of distinct *Computation/Transportation Nodes* should be precisely related: the schedule of the "next" *CN* should be that of the "previous" *CN* shifted one slot right. If not, then extra *Fractional Registers* need to be inserted just after the regular register already set between "previous" and "next" *nodes*. This *FR* should delay discriminatingly some tokens (but not all).

We shall introduce a formal model of our *FR* in the next SubSection. The block diagram of its interfaces are displayed in Figure 8.

We conjecture that, after integer latency equalization, such elements are only required just before *Computation Nodes* to where cycles with different original rates re-converge. We prove in subsection 4.4 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 their need. The intention is that a regular register coupled with a *FR* one should almost amount to a *RS*, with the only difference that the backpressure control $stop\_\{in/out\}$ signal mechanisms could be simplified due to static scheduling information computed previously.

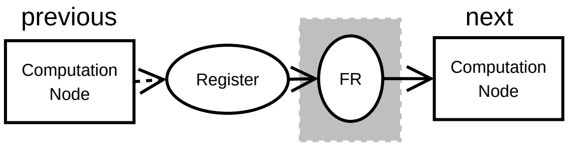

Figure 8: *Fractional Register* insertion in the Network.

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 *Transportation Node* alone (the one on the backward up arc) in a first step. This modification allows the graph to reach 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 becomes a challenge. One needs to solve two questions: first, how to generate efficiently states reachable in an *asynchronous* fashion (instead of the deterministic *asap* single successor state); second, how to discover very early that a state may be part of a periodic regime? These issues are still open. We are currently experimenting with KPASSA on efficient representation of *asynchronous* firings and resulting state spaces.

Remark 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.

4.3 Fractional Register element (FR)

We now formally describe the specific *FR*, both as a synchronous circuit in Figure 9(b) and as a corresponding syncchart (in Mealy FSM style) in Figure 9(a).

The *FR* interface consists of two input wires *val_in* and *hold*, and one output wire *val_out*. Its internal state consists of a register *catch_reg*. The register will be used to "kidnap" the valid data (and its value in a real setting) for one clock cycle whenever *hold* holds. We note $pre(catch\_reg)$ 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 data is held several instants in a row. But meanwhile there should be no new data arriving, as the *FR* can store only one value; otherwise this would cause a conflict.

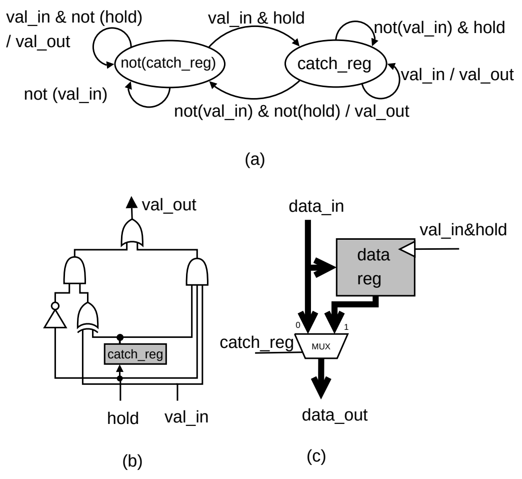

Figure 9: (a) The syncchart, (b) the interface block-diagram of the *FR*, and (c) the datapath

It is also possible that a full sequence of consecutive datas are held back one instant each in a burst fashion. But then each data/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 $hold \wedge pre(catch\_reg)$ holds then either val_in holds, in which case the new data enters and the current one leaves (by scheduling consistency the *Computation Nodes* that consumes it should then be active), or val_in does not hold, in which case the current data remains (and, again by scheduling consistency, then the *Computation Node* should be inactive). Furthermore the two extra conditions are requested:

[$hold \Rightarrow (val\_in \vee pre(catch\_reg))$:] if nothing can be held, the scheduling does not attempt to;

[$(val\_in \wedge pre(catch\_reg)) \Rightarrow hold$:] otherwise the two datas could cross the element and be output simultaneously.

The *FR* behavior amounts to the two equations:

[$catch\_reg = hold$:] the register slot is used only when the scheduling demands;

[$val\_out = val\_out_1 \vee val\_out_2$:]

- $val\_out_1 = val\_in \oplus pre(catch\_reg) \wedge \neg hold.$
- $val\_out_2 = val\_in \wedge pre(catch\_reg) \wedge hold.$

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. Its schedule should be the difference between the schedule of its source *(Computation or Transportation) Node* shifted by one instant, and the schedule of its target node; indeed, a token must be held when the target node does not fire while the source *CN* did fire to produce a token last instant, or if the token was already held at last instant.

Consider again Figure 8, we shall name w the schedule of the *previous* source *CN*, and w' the schedule of the *next* target *CN*. After the regular register delay the datas are produce to the *FR* entry on schedule $0.w$ (shifted one slot/instant right). The *Fractional Register* should hold the data exactly when the k^{th} active step at this entry is *not* the k^{th} activity step at its target *CN* that must consume it. In other words the *FR* 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 *CN* 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 *CN*, not caused by the propagation of tokens from the source *CN*, that would have to be removed.

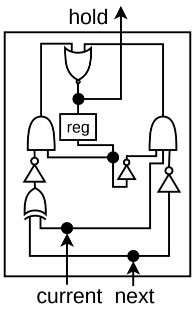

Figure 10: $hold$ implementation.

Figure 10 shows a possible implementation computing $hold$ from signals that would explicit provide the target and source schedules as inputs.

Correctness properties It can be formally proved that, under proper assumptions, a full *RS* is sequentially equivalent to a system made of a regular register followed by a fractional one, with the respective *stop_out* and *hold* signal equated (as in figure 11). The exact assumption is that a *stop_out/hold* signal is never received when the systems considered are already full (both registers occupied in each case). Providing this assumption to a model-checker is cumbersome, as it deals with internal states. It can thus be replaced by the fact that never in history there are more than one *val_in* signal received in excess of the *val_out* signals sent. This can easily be encoded by a synchronous observer.

In essence the previous property states that the two systems are equivalent *safe* for the emission of *stop_in* on a full *RS*. This emission can also be shown to be simulated by inserting the previous *HOLD* component with proper inputs. Of course this *does not* mean that the implementation will use such a dynamic *HOLD* pattern, but that simulating its effect (because the static scheduling instructs us of when to generate the signal) would make things equal to the former *RS* case.

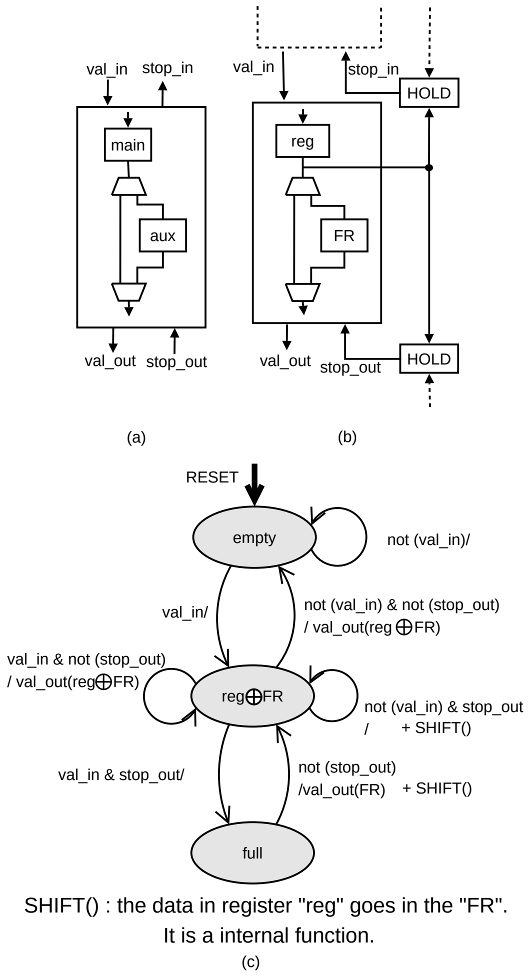

Figure 11: Equivalence of RS and FR roles

4.4 Issues of optimal FR allocation

As already mentioned in the case of a *SCC* 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 cycle(s) 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 *FRs*).

Definition 13 (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.

Conjecture 1. *If all* Computation Node *schedules are smooth, rates can be equalized using* FR *only at* Computation Node *entry points.*

Counter example 1. *We originally thought that the conjecture 1 should be sufficient, but the counter example of the figure 12 was found: Assume a simple graph formed with two cycles sharing one* CN. *The first critical cycle has 7 tokens and 11 latencies, the second one has 5 tokens and 7 latencies. It exists a stationary phase were the schedule of all* CN *is smooth (it's [10101010111] or any rotation of this word) but we need two successive* FR*s on the non critical cycle because only one* FR *should overflow.*

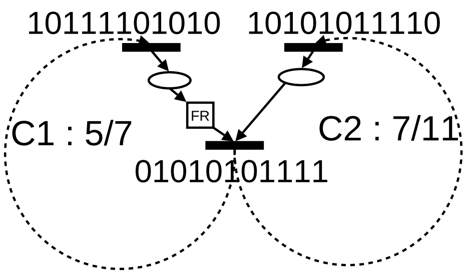

Figure 12: Counter example of the conjecture 1. the FR overflow at instant 7.

The reason of this failure is that the definition of smoothness is not restrictive enough. In the schedule of the counter-example 12, the pattern 10 is repeated 3 times at the beginning and we have 3 occurrences of 1 (which are not followed by any 0) at the end. 0 and 1 are not spread regularly enough in the schedule. However, if the schedule of the *CN* become (01011011011), we now need only one *FR* on the non critical cycle.

We propose a **new** definition:

Definition 14 (Extended Smoothness). A schedule w is said *extended smooth* if any subword, with a length l, contains either n bits at 1 or $n+1$ bits at 1, where n is equal to $\lfloor \frac{l*|w|_1}{|w|} \rfloor$, $|w|_1$ is the number of occurrences of 1 in w and $|w|$ is the length of w.

4.5 Tool implementation

Our KPASSA tool implements the various algorithmic stages described above. Given that we could not yet prove that *FR* were only required at specific locations, the tool is ready to insert some anywhere. KPASSA computes and displays the system throughput, shows critical cycles and the locations of choice for extra integer latency insertions in non-critical cycles. It then computes an explicit schedule for each *Computation and Transportation Node* (in the future it could be helpful to display only the important ones), and provides locations for *Fractional Registers* insertion. It also provides log information on the numbers of elements added, and whether perfect integer equalization was achieved in the early steps.

In the future, we plan to experiment with algorithms for finding efficient asynchronous transitory initial phases that may reach the stationary periodic regime faster than with the current *ASAP* synchronous firing rule.

Figure 13 displays a screen copy of KPASSA on a case study drawn from [9]. Using the original latency specifications our tool found a static schedule using less resources than the former implementation based on *Relay-Stations* and dynamic backpressure mechanisms. And now the activation periods of components are fully predictable.

5 Experiments on case studies

Tables 1 and 2 display benchmark results obtained with KPASSA on a number of case studies. The first examples were built from [9] for MPEG2 Video Encoder and from existing and publicly available models of structural IP block diagrams (IP MegaStore of Altera). But the latency figures were suggested by ours industrial partners of PACA CIM initiative In [11] the authors use a public-domain floorplanner to synthesize approximate latency figures, based on wire lengths induced by the placement of IPs. The last two examples are based on graph shapes and latency distribution that are *a priori* adverse to the approach (without being formerly worst-cases).

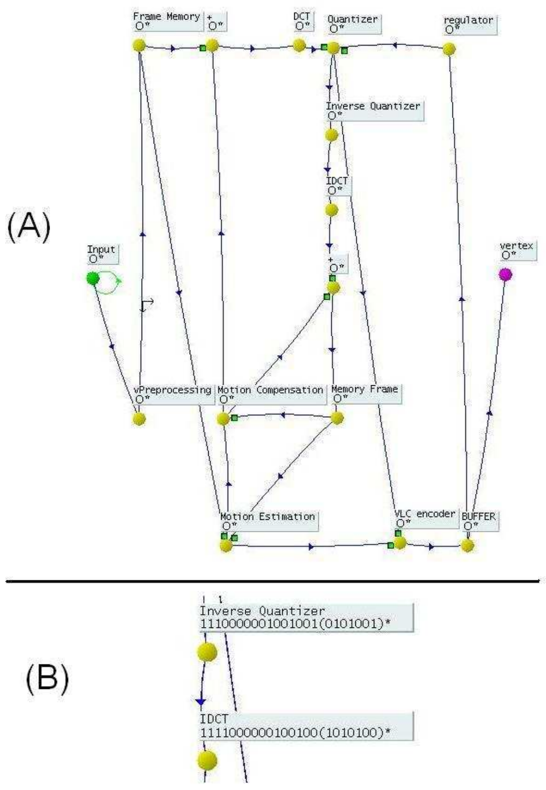

Figure 13: An example simulation result (MPEG2 Encoder) with KPASSA. In (A), the graph; in (B), the displayed schedules for two vertices

Table 1 provides features of size that are relevant to the algorithmic complexity. Table 2 reports the results obtained, about: whether Perfect Equalization holds; on the number of *Fractional Registers* required in the initial and periodic phases (note that some *FR* elements may still be needed for the initial part even in perfectly equalized cases); on the number of integer latencies added; on time and space performances.

The current implementation of the tool is not yet optimized for complexity in time and space, until now this is not yet important. The graph state encoding is naive, and algorithms are not optimal.

KPASSA is a formal tool that is able to compute effectively the length of initialization and periodic pattern, to compute an upper-bound of the number of resources used for the implementation. The tool provides huge preliminary implementations for the static-scheduled LID, but it let us experiment new ideas to optimize those implementations.

In addition to the results shown in the Tables 1 and 2, KPASSA also provides synthetic information on the criticality of Nodes: cycles can be ordered by their rates, and then Nodes by the slowest rate of a cycle it belongs to. Then the nodes are painted from red ("Hotspot") to blue ("Coldspot") accordingly. This visual information is particularly useful before *Equalization*.

6 Further Topics

Concerning the static scheduling, a number of important topics are left open for further theoretical developments:

- Relaxing the firing rule: So far the theory developed here only consider the case where local synchronous components all consume and produce token on all input and output channels in each computation step, and where they all run on the same clock. In this favorable case functional determinacy and confluence are guaranteed, with latencies only impacting the relative ordering of behaviors. So it can be proved that the relaxed-synchronous version produces the same output streams from the same input streams as the fully synchronous specification (indeed the rank of a token in a stream corresponds to its time in the synchronous model, thereby reconstructing the structure of successive instants). Several papers considered extensions in the context of GALS systems, but then ignored the issue of functional correspondence with an initial well-clocked specification, which is our important correctness criterion. This relaxation may help minimize some metrics :
  - We certainly would like to establish that *FR* are needed only at *Computation Nodes*, minimizing their number rather intuitively;
  - Discovering short and efficient (minimizing number of *FR*) 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"* that is to say flatten the power peaks. It could be achieved by some sort of retiming/recycling techniques and schedules exploration still using a relaxed firing rule;
- *Marked Graphs* do not allow control-flow (and control *modes*). The reason is, in general case such as full Petri Nets, it can no longer be asserted that token are consumed and produced at

	#Nodes	#Cycles	#Critical Cycles	Max Cycle Latency	Throughput
MPEG2 Video Encoder	16	7	3	21	3/7
Encoder MultiStandard ADPCM	12	23	23	14	1/2
H264/AVC Encoder	20	12	3	27	4/9
29116a 16bits CAST MicroCPU	11	7	3	35	3/35
Abstract Stress Cycles	40	2295	1	1054	4/29
Abstract Stress Nodes	175	3784	1	1902	4/29

Table 1: Example sizes before equalization

	Perfect Eqn.	#FR init/periodic	#Added latencies	Time	Memory
MPEG2 Video Encoder	N	9/5	18	<1 sec	˜11MB
Encoder MultiStandard ADPCM	Y	24/0	91	<1 sec	˜11MB
H264/AVC Encoder	N	18/11	0	˜ 1sec	˜11MB
29116a 16bits CAST MicroCPU	Y	0/0	0	˜ 1sec	˜11MB
Abstract Stress Cycles	N	55/24	1577	˜17 sec	˜16MB
Abstract Stress Nodes	N	59/23	2688	˜4 min	˜43MB

Table 2: Equalization performances and results (Run on P4 3.4Ghz, 1GB RAM , Linux 2.6 and JDK 1.5)

the same rate. But explicit *"branch schedules"* could maybe help regulate the branching control parts similarly to the way they control the flow rate;

Finally, the goal would be to define a general GALS modeling framework, where GALS components cold be put in GALS networks (to this day the framework is not compositional in the sense that local components need to be synchronous). A system would consist again of computation and interconnect communication blocks, this time each with appropriate triggering clocks, and of a scheduler providing the subclocks computation mechanism, based on their outer main clock and several signals carrying information on control flow.

Summary In this article we first introduced full formal models of Relay Stations and Shell Wrappers, the basic components for the theory of Latency-Insensitive Design. Altogether they allow to build a dynamic scheduling scheme which stalls traveling values in case of congestion ahead. We established a number of correctness properties holding between (lines of) RSs and SWs.

Then, using former results from scheduling theory we recognized the existence of a static periodic schedules for networks with fixed constant latencies. We tried to use these results to compute and optimize the allocation of buffering resources to the system. By *equalization* we obtain location where a full extra latency is *always* mandatory (these virtual latencies can later be absorbed in the redesign of more relaxed IP components). Fractional latencies still need to be inserted to provide *perfect* equalization of throughputs. By simulation we compute the exact schedules of Computation Nodes, and deduce the locations of Fractional Register assignments to support that. We conjectured that under simple "smoothness" assumptions on the token values distribution along graph cycles the FR elements could be inserted in an optimized fashion. We also proved properties on FR implementation, and its relation to RSs.

Finally we described a prototype implementation of the techniques used to compute schedules and allocate integer and fractional latencies to a system, together with preliminary benchmarks on several case studies.

References

[1] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications, Lille, France*, pages 19–29, July 1996. Also available as I3S technical report RR 95-52.

[2] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity: an algebra for discrete event systems*. John Wiley & Sons, 1992.

[3] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Hallbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. In *IEEE Proceedings*, number 1, pages 64–83. INRIA and IRISA/INRIA and Columbia University and Verimag/CNRS, January 2003.

[4] Julien Boucaron, Jean-Vivien Millo, and Robert De Simone. Another glance at relay stations in latency-insensitive design.

Electr. Notes Theor. Comput. Sci., vol. 146(no. 2):pages 41–59, 2006.

[5] François R. Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal design of synchronous circuits using software pipelining techniques. In *ICCD'98*, pages 62–67. IEEE Computer Society, 1998.

[6] Jacques Carlier and Philippe Chrétienne. *Problème d'ordonnancement: modélisation, complexité, algorithmes*. Masson, Paris, 1988.

[7] Luca Carloni, Kenneth McMillan, and Alberto Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20(no. 9):pp. 1059–1076, 2001.

[8] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 309–315, Piscataway, NJ, USA, 1999. IEEE Press.

[9] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency-insensitive systems. In *The Proceedings of the Design Automation Conference*, pages 361–367. UC Berkeley, June 2000.

[10] Mario R. Casu and Luca Macchiarulo. A detailed implementation of latency insensitive protocols. In *Proceedings of Formal Methods for Globally Asyncronous Locally Syncronous Architectures*, pages 94 – 103, 2003.

[11] Mario R. Casu and Luca Macchiarulo. Floorplanning for throughput. In *ISPD '04: Proceedings of the 2004 international symposium on Physical design*, pages 62–69, New York, NY, USA, 2004. ACM Press.

[12] Mario R. Casu and Luca Macchiarulo. A new approach to latency insensitive design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 576–581, New York, NY, USA, 2004. ACM Press.

[13] Ajanta Chakraborty and Mark R. Greenstreet. A minimalist source-synchronous interface. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, pages 443–447, September 2002.

[14] Tiberiu Chelcea and Steven M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Design Automation Conference*, pages 21–26, 2001.

[15] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–193, New York, NY, USA, 2006. ACM Press.

[16] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, October 1971.

[17] Ali 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.

[18] Vincent Van Dongen, Guang R. Gao, and Qi Ning. A polynomial time method for optimal software pipelining. In *CONPAR '92/ VAPP V: Proceedings of the Second Joint International Conference on Vector and Parallel Processing*, pages 613–624, London, UK, 1992. Springer-Verlag.

[19] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, MIT, Cambridge, Massachusetts, USA, September 1973.

[20] Alexandre V. Yakovlev, Albert M. Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design and Test of Computers*, vol.12(no. 1):pp. 32–40, 1995.