Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 146 (2006) 61–80

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Role of Back-Pressure in Implementing Latency-Insensitive Systems

Luca P. Carloni

Department of Computer Science
Columbia University in the City of New York, New York, NY

Abstract

Back-pressure is a logical mechanism to control the flow of information on a communication channel of a latency-insensitive system (LIS) while guaranteeing that no packet is lost. Back-pressure is necessary for building *open* LISs and it represents an interesting design alternative also for *closed* LISs because it makes possible to realize highly modular implementations with more predictable features in terms of design overhead (area, power). In discussing the role of back-pressure, we revisit the logic of the necessary building blocks, and explain the impact of the system topology on the system performance.

Keywords: Latency-Insensitive Design, GALS, marked graphs, systems-on-chip (SOC), correct-by-construction methods.

1 Introduction

The theory of latency-insensitive design is the foundation of a correct-by-construction methodology to design complex digital systems by assembling pre-designed modules [8,9]. The modules interact by exchanging data on a communication architecture that is made of point-to-point lossless FIFO channels and works based on a latency-insensitive protocol. The protocol guarantees a correct system behaves independently from the latencies of the communication channels. An important application of the theory is the latency-insensitive design methodology to build gigascale systems-on-chip (SOCs) with nanometer technologies [7]. Here latency-insensitive design provides various

[1] Email: luca@cs.columbia.edu

doi:10.1016/j.entcs.2005.05.036

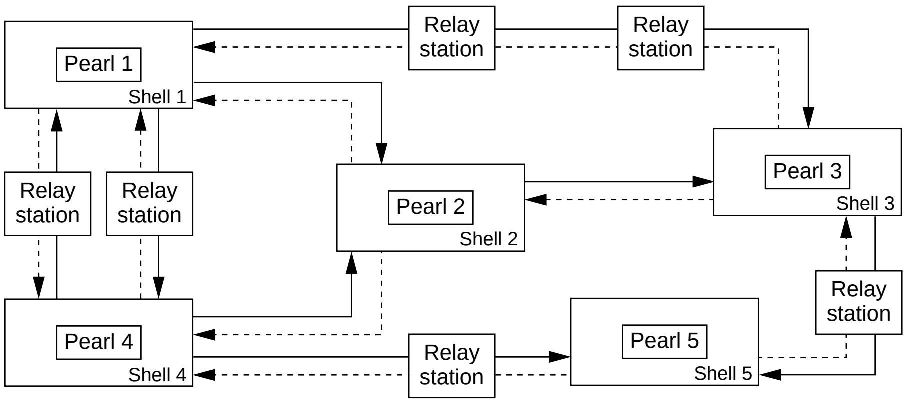

Figure 1. Shell encapsulation, relay station insertion and channel back-pressure.

practical advantages: (1) it simplifies the reuse of pre-designed and pre-verified intellectual property (IP) cores, that, as long as they are stallable, can be interfaced with the communication protocol without changing their internal structure, (2) it enables the *a-posteriori* automatic pipelining of long wires through the insertion of any number of sequential repeaters (*relay stations*), and (3) it facilitates the exploration of communication versus computation trade-offs up to late stages of the design process. Furthermore, latency-insensitive design does not require engineers to undertake a revolution in their practices: since it is based on the synchronous paradigm [1], it represents a theoretically sound framework from which to develop a new class of design flows for nanometer design through the use of traditional CAD tools.

Figure 1 illustrates the typical structure of a LIS implementation together with its main "characters". In this simple example five pre-designed modules (the *pearls* or *cores*) are encapsulated within as many interface logic blocks (the *shells*) and communicate by means of eight point-to-point *channels*. Some of the channels have been pipelined through the insertion of six relay stations. The data travelling on the channels of a LIS are distinguished in *true packet* or *void packets*. The packets are processed by shell/core pairs according to AND-causality rule: if a new true packet is available on each input channel then the core consumes a packet from each channel, thereby updating its internal state and producing a new true packet on each output channel; instead, the absence of a true packet (and, correspondingly, the presence of a void packet) on a single input channel is enough to force the shell to (1) stall the core, (2) store those true packets that are possibly present on the other channels in its input queues for later use, and (3) emit a void packet on each output channel.

In the particular implementation of Figure 1, each channel presents a counter-flow signal (represented by the dashed arrow) that is necessary to implement *back-pressure*. Back-pressure is a logic mechanism thereby on a given channel the down-link shell can request the up-link shell to temporarily stop its production of true packets [7]. The reason for this request is that the buffering capability of a shell in terms of input queues is finite. Hence, in order to avoid loss of true packets while being forced to stall its core for many consecutive cycles, a shell has ultimately to activate back-pressure on the input channels corresponding to those queues that are getting full. The cause of many consecutive stalling cycles is either back-pressure coming from output channels or the sustained lack of alignment of corresponding true packets across input channels (or, possibly, a combination of the two).

Since infinite-length queues are not physically realizable, one could argue that back-pressure is always an essential implementation feature of any LIS. However, this is not necessarily the case. In fact, in order to derive the final implementation of a LIS, designers may sometime have a choice on whether to use back-pressure or an alternative implementation style that still does not require infinite queues. The existence of this choice depends on the *nature* of the system under design, i.e. whether the system is open or closed. Further, when the choice is present, the final decision must be taken after considering the interplay between the computational structure of the system and the different design overheads introduced by the alternative solutions. In the rest of the paper we further discuss this question and we explain under which conditions this choice does not impact the system performance. While doing so we revisit a reference implementation of LIS based on back-pressure.

2 The Building Blocks of a Latency-Insensitive System with Back-Pressure

Regardless of the particular implementation style, a latency-insensitive system is made of shells encapsulating cores, channels connecting shells, and relay stations pipelining channels. In this section we revisit a reference implementation with back-pressure (first presented in [7]) and we provide more details on the logic of these building blocks.

Channels and Back-Pressure. Channels are point-to-point unidirectional links that connect a *source* shell/core pair to a *sink* shell/core pair. Data are transmitted on a channel by means of *packets*: a packet consists of a variable number of fields. In a basic reference implementation, a packet is made of two fields: *payload*, which contains the transmitted data, and *void flag*, which is a one-bit signal that, if set to 1, denotes that no data are present in the packet (*void packet*). If a packet does contain "meaningful" payload data (i.e., void is set to 0) it is called a *true packet*. In an implementation

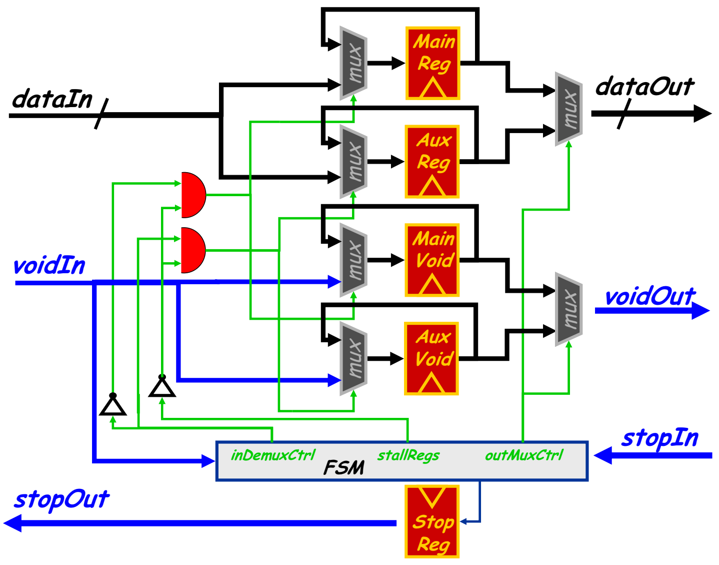

Figure 2. Schematic circuit diagram of a relay station.

with back-pressure, a *stop flag* signal travels on a channel in backward direction from the sink to the source. In practical hardware design, a channel can be implemented as a set of wires: as many wires as they are necessary to encode the data to transmit plus two additional wires for the void flag and the stop flag respectively. Channels can be pipelined as much as they need to be by distributing a finite number of relay stations on them. Besides providing a systematic method to perform wire pipelining, the insertion of relay stations on a channel creates a sort of *distributed queue*. Further, the control logic of the queue is also distributed as it is implemented by the back-pressure mechanism, which is inherently modular. Due to the increasingly distributed nature of SOCs [41], distributed queues represent a design solution more promising than having long, centralized communication queues located next to each IP core module.

Relay Stations. A diagram for a possible RTL circuit implementation of a relay station was presented in [7,11]. Figure 2 contains a more detailed description of this implementation. This circuit targets single-clock synchronous integrated circuits: at each clock cycle t a relay station takes a packet $packetIn^t$ and a stop flag $stopIn^t$ as inputs and emits a packet $packetOut^{t+1}$ and a stop flag $stopOut^{t+1}$ as outputs. Packet $packetIn^t$ contains the payload $dataIn^t$ and the void flag $voidIn^t$. Packet $packetOut^{t+1}$ has the same structure with payload $dataOut^{t+1}$ and void flag $dataOut^{t+1}$. Stop flag signals

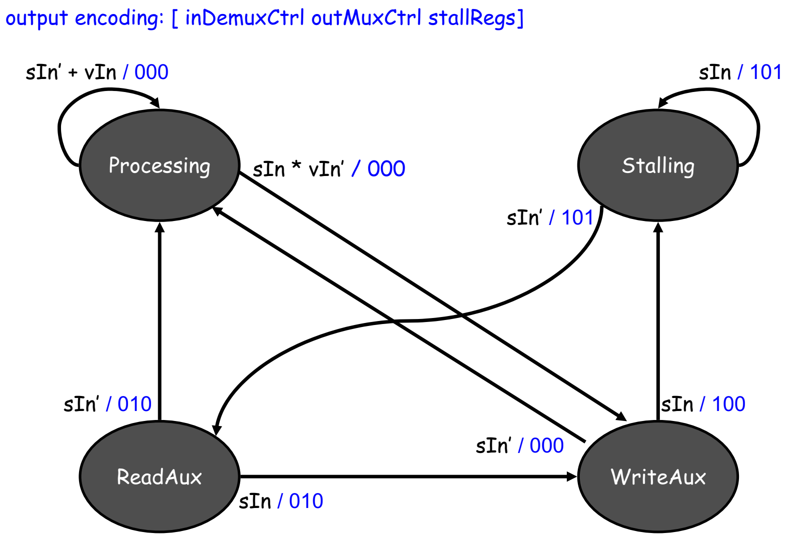

Figure 3. State-transition diagram for the FSM controlling a relay station.

$stopIn^t$ and $stopIn^{t+1}$ support the back-pressure mechanism. A key property of a relay station is that no combinational path must exist between its inputs and its outputs. Consequently, it takes a clock cycle also for the stop flag signals to propagate across the relay station in backward direction. Since a LIS is made of lossless channels, we must avoid losing data during the one cycle that is necessary to propagate the stop signal. This fact, combined with the goal of supporting a best-case maximum communication throughput (equal to one), implies that a relay station must have double storage capacity. Compared with a simple edge-triggered flip-flop, which can be similarly used to pipeline channels without backpressure, a relay station presents the characteristic twofold buffering capability (together with the necessary control logic): a *secondary* (or *auxiliary*) register is coupled to a *main* register.

As illustrated in Figure 2, three internal signals control the flow of data across a relay station: $inDemuxCtrl$, $stallRegs$, and $outMuxCtrl$. When set to 1, signal $stallRegs$ makes sure that the packets stored in both the main and the auxiliary registers are not overwritten. When $stallRegs$ is equal to 0, instead, either the main or the auxiliary registers are updated with the incoming packet based on the value of signal $inDemuxCtrl$. Signal $outMuxCtrl$ controls the output multiplexer in order to decided whether the output channel is driven by the main registers or the auxiliary registers.

Figure 3 contains the state transition diagram of a finite state machine (FSM) representing a possible detailed specification of the control logic of the relay station that refines the one sketched in [7,11]. The FSM has 2 input signals ($stopIn$, $voidIn$), 3 output signals ($inDemuxCtrl$, $outMuxCtrl$, $stallRegs$), and 4 states ($Processing$, $WriteAux$, $Stalling$, $ReadAux$). Output signals $outMuxCtrl$ and $stallRegs$ depend only on the FSM present state while output signal $inDemuxCtrl$ depends both on the FSM present state and the input signal $stopIn$. Hence, $outMuxCtrl$ and $stallRegs$ are "Moore-type outputs" while $inDemuxCtrl$ is a "Mealy-type output". However, notice that, as it is required, there are no combinational paths from the primary inputs to the primary outputs of the relay station because the outputs of the input multiplexers are sampled by either the main registers or the auxiliary registers. Notice that if the FSM is in the *Processing* state and $voidIn^t = 1$, it remains in the processing state regardless of the value of $stopIn^t$. The given FSM description doesn't include the logic to process the value of $stopOut^{t+1}$, which is simply equal to the value of $stopIn^t$, unless the relay station is in the *Processing* state and $voidIn^t = 1$: in this case $stopOut^{t+1}$ is set equal to 0. Finally, notice the following characteristic of the latency-insensitive protocol implemented by this relay station: if the *stopIn* signal is kept high by a down-link module (either a relay station or a shell) for only one clock cycle, then the relay station does not really stall (i.e. no packet is kept on the output port for more than one cycle). Conversely, the relay station knows that the down-link module is correctly sampling the packet present on its output port at a given time t when the following logic condition is satisfied: $\big(stopIn^t = 0\big) \vee \big(stopIn^t = 1 \wedge stopIn^{t-1} = 0\big)$.

Shell Encapsulation. Given a particular core module M, an instance of a shell module can be automatically synthesized as a wrapper to encapsulate M and interface it with the channels so that their combination becomes a patient system, i.e., informally, a system that "understands" the latency-insensitive protocol. The theory of latency-insensitive design guarantees that the only necessary precondition is that M be stallable [9]: at each clock cycle the internal computation of the core must be *fired* only if all inputs have arrived. In other words, the computation of M can occur at a given clock cycle only when the corresponding true packet has arrived on each input channel. The absence of a true packet is explicitly expressed by the arrival of a void packet, i.e. a packet with the void flag set to 1. Guaranteeing this *input synchronization* is the first task of the shell of a core module and it is strictly combined with *core stalling*. The shell stalls the core either because a true packet is missing on an input channel or because there is at least one output

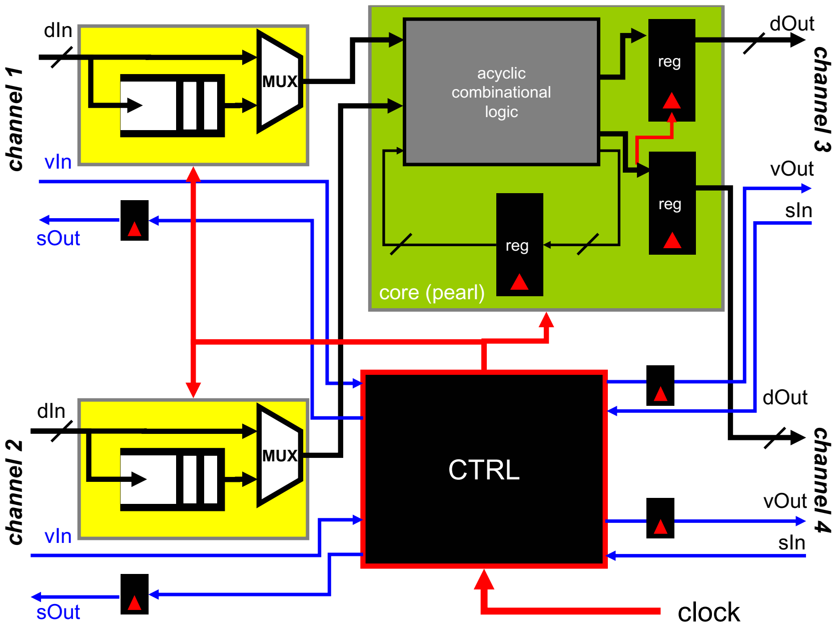

Figure 4. Shell encapsulation: making an IP core patient.

channel where the down-link (sink) shell has been keeping high the stop flag for two or more consecutive cycles (via back-pressure). While stalling the core, the shell needs to perform *input buffering* and *output propagation*. With input buffering the shell avoids losing those true packets that have regularly arrived on a given channel (and that cannot be processed by the core yet) by storing them in a dedicated queue. Output propagation, instead, is the mechanism to guarantee that a true packet is kept for as long as necessary on the output port of a channel that has requested so via back-pressure. On the other hand, during core stalling the shell must use the void flag to invalidate the packet on each output port of a channel that has not made such request. Not doing so would potentially lead to a computational error due to multiple sampling of the same packet. As it implements the protocol outlined above, the combination shell-core operates according to an AND-causality semantics [23], like a transition of an ordinary marked graph [17] or an actor of a homogeneous synchronous data flow [31]. These models of computation present several useful properties, including the possibility of computing efficiently and exactly the performance of the overall system.

Figure 4 shows the conceptual diagram of a RTL implementation of a shell encapsulating a core module with two input channels and two output channels. The control logic implements the various tasks described above. The stalling/firing mechanism is simply obtained by *gating the clock signal*, which controls the register of the core. [2] Notice that all the output signals are properly latched: for the payload field of the output packets the shell relies on the output latches of the stallable core module [3] while the auxiliary signals, i.e. the void flags and stop flags, are latched by the shell itself. Finally, notice the presence of by-passable queues on the input channels. The reasons to make them by-passable is to guarantee that if a new set of incoming true packets is available and no stalling requests came from the output channels then a new set of outgoing true packets is produced in exactly one clock cycle, i.e. without adding any cycle to the original latency of the core.

The design overhead due to shell encapsulation generally depends on the size of the core. Naturally, the larger is the core, the smaller is the relative impact of shell encapsulation. Further, to develop a library of optimized shells is a feasible task because they can generally be reused across many different core modules. In fact, the control logic of the shell remains the same regardless of the internal complexity of the core and the number of shell queues only depends on the core's I/O interface. This important advantage of the proposed latency-insensitive protocol not only simplifies the design of the shell but it also guarantees its broad applicability: as long as the core module is a stallable sequential circuit it is not necessary to know its internal structure or behavior. In other words, latency-insensitive design can be applied to any stallable *"black-box"* IP core.

3 Performance Analysis of Latency Insensitive Systems

No matter how many relay stations are introduced on the channels of a latency-insensitive system, its functional correctness is guaranteed to be preserved: the system may produce more void packets on the output channels as well as exercise more back-pressure on the input channels but, nevertheless, its processing activity progresses without deadlocking. Naturally, however, the effectiveness of latency-insensitive design is strongly related to the ability of maintaining a sufficient performance in the presence of increased channel latencies.

Nominal versus Effective Clock Frequency. In order to correctly evaluate the performance of a latency-insensitive system it is necessary to

[2] Recall that *clock gating* is a common technique in low-power hardware design.

[3] The core module of Figure 4 is a Moore finite state machine, but it could be replaced by any stallable sequential module as long as it does not have any direct combinational path from its inputs to its outputs. Hence, any arbitrarily complex pipelined circuit can be similarly encapsulated within a shell.

check how frequently it produces void packages at its output ports. Accordingly, the *throughput* $\theta(S)$ of a latency-insensitive system S is defined as the number of true packets produced by S in a given time interval. This of course corresponds to the ratio of true packets over the sum of true packets plus void packets (as observed at the system outputs during such interval) and it is a number between zero and one. It follows that, given a nominal clock frequency ϕ, the *effective frequency* of a latency-insensitive system S is

$$\phi_{eff}(S) = \phi \cdot \theta(S)$$

Throughput $\theta(S)$ depends on two factors: the internal structure of S and the interaction with the environment E where S operates.

Origin of Void Packets. A latency-insensitive system S may receive void packets at its primary inputs from the environment E in which it operates as well as generate them itself. In the first case, obviously, E impacts the throughput $\theta(S)$ of S. The second case is more interesting from a design perspective because it sets a limit on the maximum throughput that S can sustain regardless of the environment in which it operates. A properly designed shell emits void packets on its output channels only as a result of being forced to stall. In fact, when the system starts-up, each core, being a sequential process, has its output registers initialized with a true packet. [4] Instead, the generation of void packets inside system S is due to the presence of relay stations. More precisely, each relay station introduces one void packet in the system. The void packet corresponds to the initialization value [5] for the storage element of the relay station. Then, based on the computation structure of S some of these void packets may either leave the system after a transitional phase or continue to cycle in it forever. In the latter case they have a negative impact on $\theta(S)$.

[4] This follows directly from the fact that, a LIS is derived from a correct strict system where every process is a strict process and no stalling events are present [9].

[5] Since a relay station is a "design correction" that is extraneous to the original system specification, its initialization value must remain transparent to the cores (while visible to other relay stations and shells) in order to make sure that it does not corrupt their internal state. In other words, the simplest way to insert additional stateful repeaters into a sequential system without changing the internal logic of its components nor jeopardizing the correctness of its overall functional behavior is to make sure that such repeaters are initialized with values that will not get processed by the components that receive them. With a latency-insensitive protocol this can be done systematically as relay stations introduce void packets and shells make sure that the cores do not see them. Observe that the initialization of relay stations cannot be generally handled with the same methods used for retiming [32,48] because their insertion can occur in arbitrary number without the restrictions given by the retiming invariant rule (*the number of storage elements that lie on any feedback path of a synchronous circuit must remain constant through retiming*) [37].

Maximum Sustainable Throughput. Since the structure of a LIS S determines the maximum throughput that S can sustain regardless of the environment E in which it operates, we are interested in defining the intrinsic fundamental performance metric of S. We call this quantity the *maximum sustainable throughput* because S effectively runs at this throughput unless E forces it to slow down by either not providing enough true packets to process or requiring it to wait via back-pressure. As discussed next, the behavior of S can be analyzed by building a marked graph model $\mathcal{MG}_S$. In particular the maximum sustainable throughput of S can be precisely derived by performing a static analysis on the structure of $\mathcal{MG}_S$ based on the following definition. *The* maximum sustainable throughput *of a marked graph model $\mathcal{MG}_S$ is defined as*

$$\vartheta(\mathcal{MG}_S) = \begin{cases} 1 & \text{if } \mathcal{MG}_S \text{ is acyclic;} \\ \min\left\{1, \ \frac{1}{\pi(\mathcal{MG}_S)}\right\} & \text{if } \mathcal{MG}_S \text{ is cyclic and} \\ & \text{strongly connected;} \\ \min_{\forall \mathcal{MG}_{SCC} \in \mathcal{MG}_S} \left\{\vartheta(\mathcal{MG}_{SCC})\right\} & \text{otherwise.} \end{cases}$$

where $\pi(\mathcal{MG}_S)$ denotes the cycle time of $\mathcal{MG}_S$.

First, since an acyclic marked graph can sustain any rate of production/consumption, it is reasonable to set its maximum sustainable throughput equal to one by definition. [6] Second, when $\mathcal{MG}_S$ is strongly connected $\vartheta(\mathcal{MG}_S)$ is equal to the reciprocal of its cycle time that is determined by any of its critical cycles. [7] Finally, when $\mathcal{MG}_S$ is cyclic with multiple strongly connected components (SCCs), then $\vartheta(\mathcal{MG}_S)$ is effectively determined by the slowest among them. In fact, if a "slower" SCC feeds a "faster" one then it implicitly reduces the throughput of the latter. Instead, if it is the faster SCC that is positioned up-link with respect to the slower then the system is not bounded and we have token accumulation in the place connecting the two SCCs. In this case, we must interpret $\vartheta(\mathcal{MG}_S)$ as a design constraint for the implementation of S, i.e. a constraint that designer must satisfy by either "slowing down" the faster SCC or "speeding up" the slower (since infinite queues cannot be realized).

[6] This confirms the intuition because a marked graph without cycles represents a pipelined system without feedback paths. Hence, for any possible initial marking there exists a finite number K of steps after which all token vacancies (stalling events) originally in the system have been ejected through its output ports and, as long as each input port continues to receive a token at each step, each place in the system contains a token.

[7] The relationship between the cycle time of a strongly-connected marked graph and the cycle metric of its critical cycles is reported in several works including [6,35,36,38,40].

Modeling Latency-Insensitive Systems with Marked Graphs. The structure and behavior of a LIS implementation can be effectively captured with a constructive modeling approach based on marked graphs. The approach is constructive in the sense that applies a one-to-one correspondences between the building blocks of a LIS and some pre-defined marked graph structures. To give a detailed description of how to model a LIS with marked graphs goes beyond the scope of this paper and we refer to [12]. [8] However, observe that a "classic LIS" implementation can be conveniently modeled with marked graphs because at the *protocol level* it operates as a deterministic system whose behavior is fully determined once void packets are distinguished from true packets in the initial state. Naturally, it is possible to introduce other latency-insensitive protocols in order to take advantage of the fact that for some core modules it is not the case that each input channel must be sampled at every clock cycle (as proposed in [42]). However, the correct design of such protocols will require more information on the core's internal structure than what a *"black-box"* IP core gives and the analysis of their performance will likely require to go beyond simple marked graphs and use instead more powerful subclasses of Petri nets.

The constructive modeling approach based on marked-graph captures both virtual implementations where shells have unlimited memory space to store interface signals (infinite queues) as well as implementations that are based on the combination of finite queues and *back-pressure*. In both cases, we can use theoretical properties of marked graphs to prove that any LIS is live by construction, to determine whether it is bounded or not, and to statically compute its maximum sustainable throughput. For the case of *infinite queues*, these models can be used to determine if the system's throughput is high enough to satisfy the performance requirement imposed by the environment as well as to determine the minimum finite size of each queue in order to guarantee correct operations (by computing the maximum number of tokens that can be present on a given place during the operation of a bounded system). Finally, the model allows us to derive a result with important practical consequences: for any given LIS we can build a *physical* implementation based on back-pressure and finite queues with length equal to two that offers the same performances as a *virtual* implementation with infinite queues.

[8] In [10] we presented a performance analysis of LISs based on ad-hoc formalism (called *lis-graphs*, i.e. latency-insensitive system graphs). Since lis-graphs turned out to be equivalent to marked graphs, in [12] we replaced them with marked graphs.

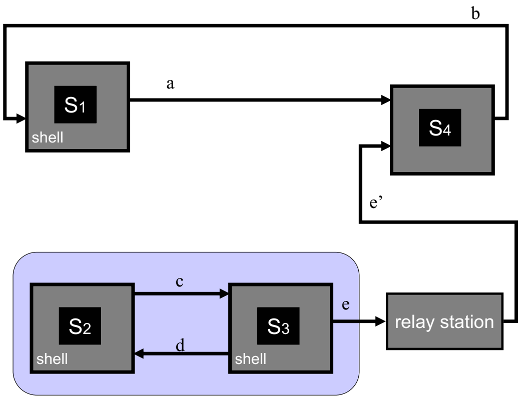

Figure 5. A non-strongly connected cyclic latency-insensitive system with a relay station inserted between its SCCs.

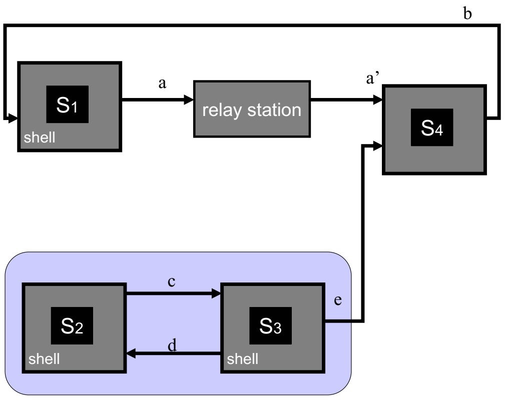

Figure 6. A non-strongly connected cyclic latency-insensitive system with a relay station inserted within a SCC.

4 Discussion

We provide here a qualitative analysis on some key aspects that should be considered while implementing a latency-insensitive system.

The Role of System Topology. As discussed in Section 3, the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$ and, ultimately, the performance of a latency-insensitive system S essentially depends on where (and in which number) the relay stations have been inserted. In this regard, we only need to analyze the computational structure of the system, which is captured by its corresponding marked graph model. We have four main scenarios:

(i) *$\mathcal{MG}_S$ is an acyclic graph.* [9] The insertion of relay stations on any channel of $\mathcal{MG}_S$ does not have impact on $\vartheta(\mathcal{MG}_S)$. Some void packets are observed at the output ports of the system, thereby forcing the corresponding true packets to arrive with some clock cycles of additional latency, but, eventually, the system reaches a steady state and no more void packets are observed. Also, no unbounded token accumulation occurs within the system. Queues of finite length are necessary in those shells that have multiple input channels and at least one of these channels receives some void packets. These channels, however, can be identified based on the location of the relay stations and the overall topology of $\vartheta(\mathcal{MG}_S)$.

(ii) *$\mathcal{MG}_S$ is a cyclic graph with several strongly connected components (SCCs) and the relay stations are inserted between them.* To analyze this scenario without loss of generality it is sufficient to consider the case of a LIS with only two SCCs as illustrated in Figure 5. The insertion of relay stations on any channel connecting two distinct SCCs of $\mathcal{MG}_S$ does not have any impact on the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$ either. This fact may appear counterintuitive. After all the down-link SCC S_{down} is a cyclic system and the shell s_4 of S_{down}, when receives a void packet from the pipelined channel, reacts producing a void packet that propagates within S_{down}. However, shell s_4 is also the shell that "retires" this void packet after it has completed a cycle around S_{down}.

(iii) *$\mathcal{MG}_S$ is a single strongly connected component.* The insertion of relay stations on any channel of $\mathcal{MG}_S$ has always a negative impact on the maximum sustainable throughput $\vartheta(\mathcal{MG}_S)$. The impact, which varies depending on the structure of the cycles of $\mathcal{MG}_S$, can be exactly calculated by computing the cycle time of $\mathcal{MG}_S$. No unbounded token accumulation occurs within the system whether the implementation is based on back-pressure or not. Unbounded token accumulation, however, can occur at the boundary between the system and the environment in which it operates.

(iv) *$\mathcal{MG}_S$ is a cyclic graph with several strongly connected components and the relay stations are inserted within some of them.* For each distinct SCC this case boils down to the previous case and the corresponding maximum sustainable throughput can be calculated exactly. The interaction

[9] Strictly speaking, if we implement an acyclic LIS using back-pressure, the final implementation cannot be acyclic (and this is reflected by its finite-queue marked graph model). In this case, however, it turns out that if we equalize the number of relay stations on all the pairs of reconvergent feedforward paths, then there is no throughput degradation. Casu and Macchiarulo made this observation in [13] where they call such optimization *path equalization*.

between the SCCs, however, needs to be managed carefully because this is the case where unbounded token accumulation can occur inside the system depending on the relative values of the maximum sustainable throughput of its SCCs. If a SCC S_{up} lies in an up-link position with respect to another SCC S_{down} and $\vartheta(S_{up}) > \vartheta(S_{down})$ then unbounded token accumulation is guaranteed to happen. For instance, Figure 5 illustrates a LIS where this scenario occurs since $\vartheta(S_{up}) = 1$ while $\vartheta(S_{down}) = \frac{2}{3}$. In order to avoid unbounded token accumulation it is necessary either to implement back-pressure or to slow-down the faster SCC. The general trade-offs between these two alternatives is discussed next.

The Role of Back-Pressure. The choice of whether to use a back-pressure mechanism in completing the physical implementation of a latency-insensitive system S that operates within an environment E is not straightforward. The result reported at the end of Section 3 guarantees that this choice does not affect the maximum sustainable throughput $\vartheta(S)$ of the system. In other words, as long as the environment is always capable of providing new true packets and does not generate ever stalling requests, back-pressure is not a factor on determining the system performance. On the other hand, back-pressure is a factor in guaranteeing that any specification of a latency-insensitive system S can be physically implemented under any possible configuration of channel pipelining. In fact, a back-pressure mechanism can always replace the need for (*unfeasible*) infinite-length queues in providing a correct, physical, system implementation that runs at the same throughput and does not experience any queue overflow. Back-pressure, however, comes with the assumption that the operational environment is ready to stall whenever the system sends back a stalling event. In this regard, the completion of a physical implementation generally boils down to a choice between two alternatives in order to handle the token accumulation problem:

(i) use back-pressure and reduce *dynamically* the token production rate of the environment E to match the maximum sustainable throughput of S;

(ii) do not use back-pressure and reduce *statically* the token production rate of E by running E with a nominal clock frequency $\phi(E)$ that matches the effective clock frequency $\phi_{eff}(S)$ of S.

This choice, however, may be restricted when S is an *open* system, i.e. a system that must be designed without being able to control also the design of the other systems that constitute its operational environment E. In this case, if E is latency-insensitive then back-pressure must be used, while if E is not latency-insensitive then the final design of S will be good only for those environments running at the effective clock frequency of S.

When S is a *closed* system, instead, the above decision is only influenced by considerations on the area and power overhead of the alternative solutions (since their performance is the same). We know that in an implementation with back-pressure it is sufficient to size the lengths of all the shell input queues to be equal to two. Therefore, back-pressure enables the creation of a very modular design. Furthermore, the area impact of each shell can be easily estimated from the information on the number of input channels of the corresponding core. In addition, it is necessary to account for the additional wires implementing the back-pressure signal and the double storage space within each relay station. The alternative is not to use back-pressure, thereby removing these additional wires and deploying a unit-capacity stateful repeater in the place of each relay station. However, in this case, it is necessary to equalize the throughput values of each SCC component in the system to avoid unbounded accumulation. Furthermore each input queue of every shell in the system must be sized *ad hoc* and its finite length may vary sensibly. [10] This is another reason why it is important to make an attempt for a balanced design, i.e. a design where communication and computation latencies are well balanced.

5 Related Work

Circuit modeling with Petri nets and marked graphs. The collection edited by Yakovlev *et al.* offers a summary of the state of the art in the application of Petri nets to the design of digital systems and circuits [50]. Ramchandani was the first to apply timed Petri nets to the analysis of asynchronous concurrent systems [39]. Williams proposed two types of marked graphs, called respectively *dependency graphs* and *folded graphs*, as a specialized model targeting the efficient computation of the exact throughput and latency for deterministic self-timed pipelines [49]. Similar approaches were proposed by Greenstreet and Steiglitz to analyze self-timed pipelines [21] and Thiele to analyze self-timed processor arrays [46]. Nielsen and Kishinevsky used unfolding and timing simulation to determine the cycle time and critical cycle of concurrent systems that can be represented as "timed signal graphs", an extension of marked graphs [36]. In order to model asynchronous circuits Burns introduced event-rule systems that are equivalent to marked graphs and defined the cycle period of an asynchronous system as the asymptotic average time separation between consecutive occurrences of the same event [5,6]. In his thesis [6], Burns discusses the problem of computing the minimum cycle

[10] The study of optimal strategies to statically balance $\phi(E)$ and $\phi_{eff}(S)$ while deriving a good floor-planning for S is a matter for further investigation.

period and explains the connection between this problem and linear programming (LP). This connection was originally observed by Magott who formulated a LP problem with $|T| + 1$ variables and $|P|$ constraints and solved it with a general-purpose polynomial algorithm [33]. Alternative formulations of this LP problem have been proposed and their relationships were discussed in [51]. Burns provides a specialized algorithm that exploits the particular structure of this LP problem to solve it in a low-order polynomial time.

The cycle time of a timed marked graph is a concept similar to the *maximum profit-to-time ratio*, for which Lawler provides an $\mathcal{O}(|V| \cdot |A| \cdot logB)$ algorithm where V is the set of vertices of G, A is the set of arcs, and B is a variable related to the desired precision of the results [30]. As discussed in [19], the maximum cycle mean problem is a special case of the *maximum profit-to-time ratio* problem. In 1978 Karp published an elegant theorem for calculating the maximum cycle mean together with a companion algorithm of complexity $\mathcal{O}(|V| \cdot |A|)$ [28]. Since then, several other algorithms have been proposed to solve the *maximum cycle mean problem* as surveyed in [19,24]. Relationships with the dual optimization problems, i.e. finding the minimum cycle mean and the minimum cost-to-time ratio, are discussed in [19].

Being ordinary Petri nets where each place has exactly one input transition and exactly one output transition, marked graphs are a model of computation equivalent to homogeneous synchronous data flows [31]. Synchronous data flows are a restricted version of data flow models of computation originally pioneered by Dennis [20] and are closely related to the *computation graphs* proposed by Karp and Miller in [29].

Works related to latency-insensitive design. Casu and Macchiarulo have proposed an alternative implementation for the building blocks of a latency-insensitive system that applies to the particular case when the computation of each core module can be scheduled statically [14]. Their implementation consists of building a shell circuit that stalls its core according to a periodic scheduling sequence, which is stored in a local shift register. Hence, such shell does not need to read the values of *void* signals and *stop* signals and these, therefore, can be removed. Also, since in this particular case there is no need for back-pressure, relay stations are replaced by normal unit-capacity stateful repeaters such as edge-triggered flip-flops. Naturally this implementation works only with closed systems.

Building on their previous work on the Polychrony design environment [22,45,47], Talpin and Le Guernic presented a process algebraic theory of behavioral type systems and applied it to the synthesis of latency-insensitive protocols. They showed that the synthesis of component wrappers can be optimized using the behavioral information carried by interface-type descriptions to yield minimized stalls and maximized throughput [44].

Latency-insensitive design has been adopted as the mode of operation for the components of "×*pipes*", a scalable and high-performance network-on-chip (NOC) architecture for both homogeneous and heterogeneous multi-processor SOCs that has been co-developed by researchers at Stanford University and the University of Bologna [2,3,18,43].

In the context of applying latency-insensitive design to the construction of NOCs Singh and Theobald have proposed "generalized latency-insensitive systems" with the extensions to: communication architecture with arbitrary topologies (i.e. beyond point-to-point channels), multi-clock systems, and "more flexible synchronous modules" where each input channel is not necessarily sampled at every clock cycle [42].

Hassoun and Alpert adopted the concept of relay stations as the basic synchronization elements in their approach to achieve simultaneous routing and buffer insertion for GALS architectures in SOC design [25,26].

Chelcea and Nowick have developed a library of robust interface circuits that makes it possible to extend the idea of latency-insensitive protocols to designs with mixed-timing domains (synchronous, asynchronous, multiple clocks) [15,16]. The library contains several low-latency, high-throughput, FIFO queues as well as two new *mixed-timing relay stations.* These circuits were designed using a modular approach: they defined a set of basic interfaces, both synchronous and asynchronous, that can be assembled to obtain a FIFO that meets the desired timing assumptions on both the senders' and receivers' end. Thus, the design of a mixed-timing FIFO is reduced to reusing and assembling a few pre-designed components. One of the important contributions of this work is precisely the novel design of relay stations for mixed asynchronous/synchronous interfaces.

The work of Jacobson *et al.* on "synchronous interlocked pipelines" [27] contains several commonalities with latency-insensitive design. The authors share some of the motivations (e.g., the dominance of interconnect delay in nanometer design) while they put a strong emphasis on the need to develop power-aware techniques that perform computation only on demand. Further, the underlying philosophy—the goal of finding *"a middle ground in techniques that can provide the benefits of asynchronous properties in a synchronous context"*—parallels the main motivation of the latency-insensitive design methodology [7] and the strategy towards this goal partially relies on similar techniques such as using clock-gating to implement fine-grained stalling and performing pipelined stalling in the backward direction (pipelined backpressure). An important difference between the two approaches lies at the implementation level, particularly in the circuitry used to make the stages of the synchronous pipeline. A relay station presents an "auxiliary register" in parallel to each "stage register" in order to avoid losing data during the single cycle necessary for the back-pressure stalling signal to cross the stage. In [27], instead, there is no insertion of parallel extra registers and the loss of data is avoided by using every other register in series. As a consequence, Jacobson *et al.* report that the maximum occupancy of the pipeline, when no stalling has occurred, is of $N/2$ data items, where N is the number of (serial) stage registers. This is equivalent to the occupancy of an analogous pipeline with N relay stations, where, however, the flow throughput is 1 instead of 0.5.

Finally Borgatti *et al.* have proposed a reconfigurable on-chip communication network that consists of a multi-context, programmable crossbar implemented using a matrix of modified Flash-EEPROM devices [4]. In order to improve the speed of the programmable interconnet, they combined the concept of *elastic interconnect* [34] with a newly-designed communication protocol. This operates similarly to a latency-insensitive system with backpressure as it guarantees lossless communication through the use of a *congestion signal*, which propagates in opposite direction with respect to the flow of data and control.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous language twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[2] D. Bertozzi and L. Benini. $\times$*pipes*: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31, 2004.

[3] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):113–129, February 2005.

[4] M. Borgatti, C. Auricchio, R. Pelliconi, R. Canegallo, C. Gazzina, A. Tosoni, and P. Rolandi. A multi-context 6.4Gb/s/channel on-chip communication network using 0.18μm flash-EEPROM switches and elastic interconnects. In *ISSCC Digest of Technical Papers*, February 2003.

[5] S. M. Burns and A. J. Martin. Performance analysis and optimization of asynchronous circuits. In *Advanced Research in VLSI*, pages 71–86. MIT Press, 1991.

[6] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1991.

[7] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for "correct-by-construction" latency insensitive design. In *Proceedings International Conference on Computer-Aided Design*, pages 309–315, San Jose, CA, November 1999. IEEE.

[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification*, volume 1633, pages 123–133, Trento, Italy, July 1999. Springer Verlag.

[9] 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*, 20(9):1059–1076, September 2001.

[10] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the Design Automation Conference*, pages 361–367, Los Angeles, CA, June 2000. IEEE.

[11] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, Sep-Oct 2002.

[12] Luca P. Carloni. *Latency-Insensitive Design*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, Berkeley, CA 94720, August 2004. Memorandum No. UCB/ERL M04/29.

[13] M. R. Casu and L. Macchiarulo. Issues in implementing latency insensitive protocols. In *Proc. of the Conf. on Design, Automation and Test in Europe*, February 2004.

[14] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proc. of the Design Automation Conf.*, pages 576–581, June 2004.

[15] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proc. of the Design Automation Conf.*, pages 21–26, 2001.

[16] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):857–873, August 2004.

[17] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Science*, pages 511–523, 1971.

[18] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. $\times$*pipes*: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *Proc. Intl. Conf. on Computer Design*, pages 536–541, October 2003.

[19] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, October 1998.

[20] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, Berlin, 1974. Springer Verlag.

[21] M. R. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2(3):139–148, November 1990.

[22] P. Le Guernic, J. P. Talpin, and J. C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12(3):261–303, April 2003.

[23] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101(2):265–288, 1992.

[24] M. Hartmann and J. B. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567–574, 1993.

[25] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(11):1580–1588, November 2003.

[26] S. Hassoun, C. J. Alpert, and M. Thiagarajan. Optimal buffered routing path constructions for single and multiple clock domain systems. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 247–253, 2002.

[27] H.M. Jacobson, P.N. Kudva, P. Bose, P.W. Cook, S.E. Schuster, E.G. Mercer, and C.J. Myers. Synchronous interlocked pipelines. In *8th IEEE International Symposium on Asynchronous Circuits and Systems*, April 2002.

[28] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978.

[29] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, 14(6):309–311, November 1966.

[30] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rhinehart and Winston, 1976.

[31] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[32] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[33] J. Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters*, 18(1):7–13, 1984.

[34] M. Mizuno, W. J. Dally, and H. Onishi. Elastic interconnects: Repeater-inserted long wiring capable of compressing and decompressing data. In *ISSCC Digest of Technical Papers*, pages 346–347, February 2001.

[35] T. Murata. Petri Nets, marked graphs and circuit-system theory. *Circuits and Systems*, 11(2):2–12, June 1977.

[36] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *Proc. ACM/IEEE Design Automation Conference*, pages 70–76, June 1994.

[37] M. C. Papaefthymiou. Understanding retiming through maximum average-weight cycles. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 338–348, 1991.

[38] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980.

[39] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., February 1974.

[40] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):309–311, October 1968.

[41] P. Saxena, N. Menezes, P. Cocchini, and D.A. Kirkpatrick. Repeater scaling and its impact on CAD. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):451–462, April 2004.

[42] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 21008–21013, February 2004.

[43] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, and G. De Micheli D. Bertozzi. $\times$*pipes* lite: A synthesis oriented design library for networks on chips. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1188–1193, March 2005.

[44] J. P. Talpin and P. Le Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. *Journal of Formal Methods in System Design*, to appear in 2005.

[45] J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Formal refinement-checking in a system-level design methodology. *Fundamenta Informaticae*, pages 243–273, July 2004.

[46] Lothar Thiele. On the analysis and optimization of self-timed processor arrays. *Integration, the VLSI Journal*, 12(2):167–187, December 1991.

[47] The POLYCHRONY Toolset. Developed at IRISA. Available at http://www.irisa.fr/espresso/Polychrony/.

[48] H. Touati and R. K. Brayton. Computing the initial states of retimed circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 157–162, January 1993.

[49] Ted E. Williams. Latency and throughput tradeoffs in self-timed asynchronous pipelines and rings. Technical Report CSL-TR-90-431, Stanford University, August 1990.

[50] Alex Yakovlev, Luis Gomes, and Luciano Lavagno (Eds.). *Hardware Design and Petri Nets.* Kluwer Academic Publishers, 2000.

[51] T. Yamada and S. Kataoka. On some LP problems for performance evaluation of marked graphs. *IEEE Transactions on Automatic Control*, 39(3):696–698, 1994.