Theory of Latency-Insensitive Design

Luca P. Carloni, *Student Member, IEEE*, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*—The theory of latency-insensitive design is presented as the foundation of a new correct-by-construction methodology to design complex systems by assembling intellectual property components. Latency-insensitive designs are synchronous distributed systems and are realized by composing functional modules that exchange data on communication channels according to an appropriate protocol. The protocol works on the assumption that the modules are stallable, a weak condition to ask them to obey. The goal of the protocol is to guarantee that latency-insensitive designs composed of functionally correct modules behave correctly independently of the channel latencies. This allows us to increase the robustness of a design implementation because any delay variations of a channel can be "recovered" by changing the channel latency while the overall system functionality remains unaffected. As a consequence, an important application of the proposed theory is represented by the latency-insensitive methodology to design large digital integrated circuits by using deep submicrometer technologies.**

***Index Terms*—Deep submicrometer design, formal methods, latency-insensitive protocols, system design.**

I. Introduction

THE THEORY of latency-insensitive design formally separates communication from computation by defining a system as a collection of computational processes that exchange data by means of communication channels. The communication is governed by an abstract protocol, whose main characteristic is to be insensitive to the latencies of the channels. The theory may be applied as a rigorous basis to design complex digital systems by simply composing predesigned and verified components so that the composition satisfies, formally and "by construction," the required properties of synchronization and communication. In particular, relevant applications may be found in the design of integrated circuits to be implemented with the future generations of process technologies. Indeed, for the so-called deep submicrometer (DSM) technologies (0.1 μm and below), where millions of gates will be customary, a design methodology that guarantees by construction that certain key properties are satisfied appears as the only hope to achieve correct designs in short time. Furthermore, the characteristics of DSM designs will exacerbate the *timing-closure problem* that is already present with the current technologies: the designers of semicustom integrated circuits are forced to iterate many times between logic synthesis and layout generation because the two steps are performed independently and synthesis uses statistical delay models, which badly estimate the postlayout wire load capacitance [1], [2].

In fact, despite the increase in number of layers and in aspect ratios, the resistance–capacitance delay of an average metal line with constant length is getting worse with each process generation [3], [4]. This effect, combined with the increases in operating frequency, die size, and average interconnect length, is making interconnect delay becoming a larger fraction of the clock-cycle time [5]. Introducing copper metallization and low-κ dielectric insulators helps reducing the interconnect delay, but these one-time improvements will not suffice in the long run as feature size continues to shrink [5], [6]. Furthermore, while the number of gates reachable in a cycle will not change significantly and the on-chip bandwidth that wires provide will continue to grow, the percentage of the die reachable within one clock cycle will decrease dramatically: we will soon reach a point where more gates can be fit on a chip than can communicate in one cycle [7]. In particular, it has been predicted that for DSM designs, a signal will need more than ten clock cycles to traverse the entire chip area [8]. Also, it has been estimated that only a fraction of the chip area between 0.4% and 1.4% will be reachable in one clock cycle [9]. Hence, limiting the on-chip distance traveled by critical signals will be the key to guarantee the performance of the overall design. However, since precise data on wire lengths are available only late in the design process, several costly redesigns become necessary to change the placement or the speed of the chip modules while satisfying performance and functionality constraints [2].

The theory of latency-insensitive design provides the foundation for a new design methodology that maintains the inherent simplicity of synchronous design and yet does not suffer of the "interconnect-delay problem." According to our approach, the system can be thought as completely synchronous, i.e., just as a collection of modules that communicate by means of channels having "zero delay," i.e., a delay negligible with respect to the period of the common clock signal (*synchronous assumption*). We refer this clock as the *virtual clock* and we call a system that is specified starting from this assumption *strict*. Once the final implementation of the system is derived, its operation is controlled by a *real clock* that has a precise frequency value. Unfortunately, due the above-mentioned DSM effects, some of the wires implementing these channels on the final layout may likely require a delay longer than one real clock cycle to transmit the appropriate signals. However, the proposed theory guarantees that it is not necessary to complete costly redesign iterations or to slow down the real clock. The key idea is borrowed from pipelining [10], [11]: partition the long wires into segments whose lengths satisfy the

Manuscript received June 2, 2000; revised April 23, 2001. This work was supported in part by the Semiconductor Research Corporation, the Consiglio Nazionale delle Ricerche, Italy, and Cadence Design Systems. This paper was recommended by Associate Editor M. Papaefthymiou.

L. P. Carloni and A. L. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 USA (e-mail: lcarloni@ic.eecs.berkeley.edu; alberto@ic.eecs.berkeley.edu).

K. L. McMillan is with Cadence Berkeley Laboratories, Berkeley, CA 94704 USA (e-mail: mcmillan@cadence.com).

Publisher Item Identifier S 0278-0070(01)06887-7.

timing requirements imposed by the real clock by inserting logic blocks called *relay stations*, which have a function similar to the one of latches on a pipelined data path. While the timing requirements imposed by the real clock are now met by construction, the latency of a channel connecting two modules may end up being two or more clock cycles. Still, if the functionality of the design is based on the sequencing of the output signals and not on their exact timing, then this modification of the design does not change its functional correctness provided that all its components are *patient processes*. Informally, a module is a patient process if its behavior does not depend on the latency of the communication channels because it is compliant with a *latency-insensitive communication protocol*. The key point of this approach is to relax time constraints during the early phases of the design when correct measures of the delay paths among the modules are not yet available. Instead, the specification of a complex system is strongly simplified if performed under the synchronous assumption. After the corresponding physical implementation is completed, if there are mismatches between the time constraints and the interconnect delays among the system modules, they can be easily fixed by inserting the right amount of relay stations. Since every module works accordingly to the latency-insensitive protocol, no modification in the functionality or the layout of the individual modules is necessary to reflect any necessary changes in wire latencies. The relevance of this approach for DSM design has been recently confirmed by the first details that Intel has unveiled on the new hyperpipelined NETBURST microarchitecture of its Pentium 4 processor. As reported in [12], NETBURST is the first pipeline containing instances of a stage dedicated exclusively to handle wire delays: in fact, a so-called *drive stage* is used only to move signal across the chip without performing any computation and, therefore, can be seen as a physical implementation of a relay station.

In this paper, we introduce these concepts formally and prove the properties outlined above. Section II discusses background and related work and, particularly, how the synchronous assumption distinguishes our design methodology from similar approaches proposed in the asynchronous design community during the past three decades. In Section III, we give the foundation of latency-insensitive design by presenting the notion of patient processes. At the core of our theory, originally presented in [13], lies the definition of latency equivalence. Two systems are latency equivalent if on every channel, they present the same data streams, i.e., the same ordered sequence of data, but, possibly, with different timing. We show that for patient processes, the notion of latency equivalence is compositional by proving the following theorems: 1) the intersection of two patient processes is a patient process; 2) given two pairs of latency equivalent patient processes, their pairwise intersections are also latency equivalent; and 3) for all pairs of strict processes P_1, P_2 and patient processes Q_1, Q_2, if P_1 is latency equivalent to Q_1 and P_2 is latency equivalent to Q_2, then their intersections are latency equivalent. As a consequence, we can derive the major result of our theory: if all processes in a strict system are replaced by corresponding patient processes, then the resulting system is patient and latency equivalent to the original one.

In Section IV, we define the notion of relay station: we illustrate its main properties and we show how in a system of patient processes, the communication channels can be segmented by introducing relay stations.

Section V discusses the assumption under which a generic strict system can be transformed into a patient one, i.e., its components must be *stallable*. We also delineate how the present theory leads to the definition of the latency-insensitive methodology for digital integrated circuit design: the methodology is centered on a simple noniterative design flow that can leverage common layout and synthesis computer-aided design (CAD) tools as well as state-of-the art formal verification techniques [14]. Naturally, the effectiveness of the latency-insensitive design methodology is strongly related to the ability of maintaining a sufficient communication throughput in the presence of increased channel latencies. This problem, which is just one instance of a more general issue that has to be faced while realizing integrated circuits with DSM process technologies, is discussed in Section V-D together with some techniques that can be used to handle it. Finally, to experiment the proposed methodology, we performed a latency-insensitive design of PDLX, an out-of-order microprocessor with speculative execution. This design experiment is discussed in Section VI.

II. Background and Related Work

The theory of latency-insensitive design is clearly reminiscent of many ideas which have been proposed in the asynchronous design community during the past three decades [15], [16]. In particular, the idea of a design methodology which is inherently modular is already present in the work by Clark and Molnar [17], [18]. To separate the design of these modules from the design of the system and make the entire process amenable to automation, the modules must be implemented as delay-insensitive circuits [19], [20]. A delay-insensitive circuit is designed to operate correctly regardless of the delays on its gates and wires (unbounded delay model) [21]. However, it has been proven that almost no useful delay-insensitive circuits can be built if one is restricted to a class of simple logic gates [22], [23]. To be able to build complex systems, one must use more complex components, which are "externally" delay insensitive, while "internally" are designed by carefully verifying their timing and avoiding or tolerating metastability [20], [24], [25]. By slightly relaxing the unbounded delay model and allowing "isochronic forks,"[1] practical *quasi-delay-insensitive* circuits can be built using simple logic gates [26]. A further relaxation leads to *speed-independent* circuits, which operate correctly regardless of gate delays, while wire delays are assumed to be negligible [27]–[29].

In 1985, Van de Sneupschet observed that the decreasing feature size of very large scale integration devices would have lead to "a decrease of the propagation speed of electrical signals relative to the switching speed," and proposed the use of suitable communication protocols to obtain chip designs whose correct operation is independent of the propagation speed [30]. This work has more than one contact point with the present paper, but differs on the basic fact that leads to the choice of speed-independent circuits over synchronous circuits. Dill has also pro-

[1]A bounded skew is allowed between the different branches of a net.

posed a trace theory for modeling and specifying speed-independent circuits that is the basis for a hierarchical verification approach [28]. Parallels with our paper can also be found in some of the ideas that have been proposed in the field of high-level synthesis to schedule the sequential execution of interacting processes under unbounded timing constraints [31], [32].

As we move toward the design of integrated circuits to be realized with DSM technologies, the delays of long intermodule wires are becoming dominant with respect to both the delays of the intramodule wires and those of the logic gates. More importantly, intermodule delays are difficult to predict or to control during the different phases of the design of a chip, leading to an exacerbation of the timing closure problem. Delay-insensitive approaches as well as the latency-insensitive methodology allow the designer to specify and implement the system while assuming that intermodule wire delays may vary arbitrarily. However, while a delay-insensitive system is based on the assumption that the delay between two subsequent events on a communication channel is completely arbitrary, in the case of a latency-insensitive system, this arbitrary delay is forced to be a *multiple of the clock period*. The key point is that this kind of *discretization* allows us to leverage well-accepted design methodologies for the design and validation of synchronous systems. In fact, the basic distinction between any of the previous asynchronous design methodologies and the latency-insensitive approach is essentially that a latency-insensitive system is specified as a synchronous system. Notice that we say "specified" because from an implementation viewpoint, a latency-insensitive communication protocol can also be realized using *hand-shaking signaling* techniques (such as request/acknowledge protocols), which are typically asynchronous.[2] However, from a specification point of view, each module (as well as the overall system) is viewed as a synchronous system relying on the previously cited *synchronous assumption*. Now, to specify a complex system as a collection of modules whose state is updated collectively in one "zero-time" step is naturally simpler than specifying the same system as the interaction of many components whose state is updated following an intricate set of interdependency relations. Furthermore, the synchronous specification allows us to modify the traditional semicustom design methodology slightly by simply inserting a step to encapsulate each synchronous module within a so-called *shell process*. Finally, the impact is very different also from a validation point of view because simulation is naturally a less complex task for a synchronous system than an equivalent asynchronous one. In conclusion, the theory of latency-insensitive design leads to a methodology that can be implemented on top of a commonly adopted design flow, while any asynchronous approach forces the designers to use new tools and, more importantly, to think of the digital system in a completely different way.

III. Latency Insensitivity

To develop our theory formally, we adopt the *tagged-signal model*, which has been proposed recently by Lee and Sangiovanni-Vincentelli, to represent complex systems as collections of signals and processes [33].

A. Tagged-Signal Model

Given a set of *values* $\mathcal{V}$ and a set of *tags* $\mathcal{T}$, an *event* is a member of $\mathcal{V} \times \mathcal{T}$. Two events are *synchronous* if they have the same tag. A *signal* s is a set of events. Two signals are synchronous if each event in one signal is synchronous with an event in the other signal and vice versa. Synchronous signals must have the same set of tags.

The set of all N-tuples of signals is denoted $\mathcal{S}^N$. A *process* P is a subset of $\mathcal{S}^N$. A particular N-tuple $\mathbf{s} \in \mathcal{S}^N$ satisfies the process if $\mathbf{s} \in P$. A N-tuple $\mathbf{s}$ that satisfies a process is called a *behavior* of the process. Thus, a process is a set of possible behaviors.[3] A *composition of processes* (also called a *system*) $\{P_1, \ldots, P_M\}$ is a new process defined as the intersection of their behaviors $P = \cap_{m=1}^{M} P_m$. Since processes can be defined over different sets of signals, we need to extend the set of signals over which each process is defined to contain all the signals of all processes to form the composition. Note that the extension changes the behavior of the processes only formally.

Let $J = (j_1, \ldots, j_h)$ be an ordered set of integers in the range $[1, N]$. The *projection* of a behavior $b = (s_1, \ldots, s_N) \in \mathcal{S}^N$ onto $\mathcal{S}^h$ is $\mathrm{proj}_J(b) = (s_{j_1}, \ldots, s_{j_h})$. The projection of a process $P \subseteq \mathcal{S}^N$ onto $\mathcal{S}^h$ is $\mathrm{proj}_J(P) = (\mathbf{s}' \mid \exists \mathbf{s} \in P \wedge \mathrm{proj}_J(\mathbf{s}) = \mathbf{s}')$. A *connection* C is a particularly simple process, where two (or more) of the signals in the N-tuple are constrained to be identical. For instance, $C(i,j,k) \subset \mathcal{S}^N : (s_1, \ldots, s_N) \in C(i,j,k) \Leftrightarrow s_i = s_j = s_k$, with $i, j, k \in [1, N]$.

An *input* to a process $P \subseteq \mathcal{S}^N$ is an externally imposed constraint $P_I \subseteq \mathcal{S}^N$ such that $P_I \cap P$ is the total set of acceptable behaviors. The set of all possible inputs $\mathcal{I} \subseteq \mathcal{S}^n$ is a further characterization of a process. Given a process P, if $\mathcal{I} = \{\mathcal{S}^N\}$, then the set of acceptable behaviors is $\{\mathcal{S}^N\} \cap P = P$ and the process does not have input constraints (the process is *closed*). Commonly, one considers processes having input signals and output signals. In this case, given process P, the set of signals can be partitioned into three disjoint subsets by partitioning the index set as $\{1, \ldots, N\} = I \cup O \cup R$, where I is the ordered set of indexes for the input signals of P, O is the ordered set of indexes for the output signals and R is the ordered set of indexes for the remaining signals (also called irrelevant signals with respect to P). A process is *functional* with respect to (I, O) if for all behaviors $b \in P$ and $b' \in P$, $\mathrm{proj}_I(b) = \mathrm{proj}_I(b')$ implies $\mathrm{proj}_O(b) = \mathrm{proj}_O(b')$. Hence, given a function $F : S^{|I|} \to S^{|O|}$, we can completely characterize a functional process P by the tuple (F, I, O). A process P is *determinate* if for any input $I \in \mathcal{I}$, then either $|I \cap P| = 1$ or $|I \cap P| = 0$. Otherwise, it is *nondeterminate*.

In a *synchronous system*, every signal in the system is synchronous with every other signal. In a *timed system*, the set $\mathcal{T}$ of tags, also called *timestamps*, is a totally ordered set. The ordering among the timestamps of a signal s induces a natural order on the set of events of s. A functional process

[2]Here, the communication bandwidth would be limited by the inverse of the longest of the round trip times between pairs of communicating relay stations.

[3]For $N \geq 2$, processes may also be viewed as a relation between the N signals in $\mathbf{s} = (s_1, \ldots, s_N)$.

$$\begin{aligned}\sigma(s_1) &= \iota_1\ \iota_2\ \iota_3\ \iota_1\ \iota_4\ \iota_1\ \iota_2\ \iota_2\ \iota_1\ \iota_3\ \tau\ \tau\ \tau\ \tau\ \tau\ \ldots\\ \sigma(s_2) &= \iota_1\ \tau\ \iota_2\ \tau\ \iota_2\ \tau\ \iota_3\ \tau\ \tau\ \iota_4\ \tau\ \iota_5\ \tau\ \iota_6\ \tau\ \iota_3\ \tau\ \tau\ \iota_1\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \tau\ \tau\ldots\end{aligned}$$

Fig. 1. Strict signal s_1 and a stalled signal s_2.

is *(strictly) causal* if two outputs can only differ at timestamps that (strictly) follow the timestamps when the inputs producing these outputs show a difference. More formally, given a metric d on the set $\mathcal{S}^N$ of N-tuples of signals,[4] we have the following: a functional process $P=(F,I,O)$ is *causal* if $\forall s_i,s_j \in S^{|I|}(d(F(s_i),F(s_j)) \le d(s_i,s_j))$; a functional process $P=(F,I,O)$ is *strictly causal* if $\forall s_i,s_j \in S^{|I|}(d(F(s_i),F(s_j)) < d(s_i,s_j))$.

B. Informative Events and Stalling Events

A *latency-insensitive system* is a synchronous timed system whose set of values $\mathcal{V}$ is equal to $\Sigma\cup\{\tau\}$, where Σ is the set of *informative symbols*, which are exchanged among modules and $\tau\notin\Sigma$ is a special symbol, representing the absence of an informative symbol. The absence of an informative symbol may result from either lack of valid data to transmit or back-pressure, i.e., a request to delay a transmission. From now on, all signals are assumed to be synchronous. The set of timestamps is assumed to be in one-to-one correspondence with the set $\mathbb{N}$ of natural numbers. An event is called *informative* if it has an informative symbol ι_i as value.[5] An event whose value is a τ symbol is said to be a *stalling event* (or τ *event*).[6]

Definition III.1: $\mathcal{E}(s)$ denotes the set of events of signal s, while $\mathcal{E}_\iota(s)$ and $\mathcal{E}_\tau(s)$ are the set of informative events and the set of stalling events of s, respectively. The kth event (v_k,t_k) of a signal s is denoted $e_k(s)$. $\mathcal{T}(s)$ denotes the set of timestamps in signal s, while $\mathcal{T}_\iota(s)$ is the set of timestamps corresponding to informative events.

Processes exchange "useful" data by sending and receiving informative events. Ideally, only informative events should be communicated among processes. However, in a latency-insensitive system, a process may not have data to output at a given timestamp, thus requiring the output of a stalling event at that timestamp. Alternatively, it may happen that a down-link process that is not ready to receive new data requests the up-link process to avoid sending them and, as a consequence, the latter reacts by emitting a stalling event (back-pressure).

Definition III.2: The set of all sequences of elements in $\Sigma\cup\{\tau\}$ is denoted by Σ_{lat}. The length of a sequence σ is $|\sigma|$ if it is finite, otherwise it is infinity. The empty sequence is denoted as ϵ and, by definition, $|\epsilon|=0$. The ith term of a sequence σ is denoted σ_i.

Definition III.3: Function $\sigma:\mathcal{S}\times\mathcal{T}^2\rightarrow\Sigma_{\text{lat}}$ takes a signal $s=\{(v_0,t_0),(v_1,t_1),\ldots\}$ and an ordered pair of timestamps $(t_i,t_j), i\le j$ and returns a sequence $\sigma_{[t_i,t_j]}\in\Sigma_{\text{lat}}$ s.t.[7] $\sigma_{[t_i,t_j]}(s)=v_i,v_{i+1},\ldots,v_j$. The sequence of values of a signal is denoted $\sigma(s)$. The infinite subsequence of values corresponding to an infinite sequence of events starting from t_i is denoted $\sigma_{[t_i,\infty]}(s)$.

For example, considering signal

$$s=\{(\iota_1,t_1),(\iota_2,t_2),(\tau,t_3),(\iota_2,t_4),(\iota_1,t_5),(\tau,t_6)\}$$

we have[8]

$$\begin{aligned}\sigma(s)&=\iota_1\ \iota_2\ \tau\ \iota_2\ \iota_1\ \tau\\ \sigma_{[t_2,t_4]}(s)&=\iota_2\ \tau\ \iota_2\\ \sigma_{[t_5,t_5]}(s)&=\iota_1\end{aligned}$$

and $|\sigma(s)|=6, |\sigma_{t_2,t_4}(s)|=3, |\sigma_{t_5,t_5}(s)|=1$, respectively.

To manipulate sequences of values, we define the following filtering operators.

Definition III.4: $\mathcal{F}_\iota:\Sigma_{\text{lat}}\rightarrow\Sigma^\star$ returns a sequence $\sigma'=\mathcal{F}_\iota[\sigma]$ s.t.

$$\sigma'_i=\begin{cases}\sigma_{[t_i,t_i]}(s), & \text{if } \sigma_{[t_i,t_i]}(s)\in\Sigma\\ \epsilon, & \text{otherwise}\end{cases}.$$

Definition III.5: $\mathcal{F}_\tau:\Sigma_{\text{lat}}\rightarrow\{\tau\}^\star$ returns a sequence $\sigma'=\mathcal{F}_\tau[\sigma]$ s.t.

$$\sigma'_i=\begin{cases}\sigma_{[t_i,t_i]}(s), & \text{if } \sigma_{[t_i,t_i]}(s)=\tau\\ \epsilon, & \text{otherwise}\end{cases}.$$

For instance, if $\sigma(s)=\iota_1\ \iota_2\ \tau\ \iota_2\ \iota_1\ \tau$, then $\mathcal{F}_\iota[\sigma(s)]=\iota_1\ \iota_2\ \iota_2\ \iota_1$ and $\mathcal{F}_\tau[\sigma(s)]=\tau\ \tau$. Obviously, $|\sigma(s)|=|\mathcal{F}_\iota[\sigma(s)]|+|\mathcal{F}_\tau[\sigma(s)]|$. Latency-insensitive systems are assumed to have a finite horizon over which informative events appear, i.e., for each signal s, there is a greatest timestamp $T\in\mathcal{T}_\iota(s)$ that corresponds to the "last" informative event. However, to build our theory, we need to extend the set of signals of a latency-insensitive system over an infinite horizon by adding a set of timestamps such that all events with timestamp greater than T have τ values.

Definition III.6: A signal s is strict if and only if all informative events precede all stalling events, i.e., iff there exists a $k\in\mathbb{N}$ s.t. $|\mathcal{F}_\tau[\sigma_{[t_0,t_k]}(s)]|=0$ and $|\mathcal{F}_\iota[\sigma_{[t_k,t_\infty]}(s)]|=0$. A signal that is not strict is said to be delayed (or stalled).

Fig. 1 illustrates the sequences associated to two signals presenting ten informative events each: s_1 is a strict signal with greatest timestamp equal to ten, while s_2 is a stalled signal with greatest timestamp equal to 21.

C. Latency Equivalence

Two signals are latency equivalent if they present the same sequence of informative events, i.e., they are identical except for different delays between two successive informative events.

[4]For instance, in [33], it is considered the Cantor metric $d(s_i,s_j)=\sup\{(1/2)^t\,|\,s_i(t)\ne s_j(t), t\in\mathcal{T}\}$.

[5]We use subscripts to distinguish among the different informative symbols of $\Sigma:\iota_1,\iota_2,\iota_3,\ldots$.

[6]The use of the τ event is similar to the role played by the *absence* symbol $\perp$ in the synchronous language SIGNAL [34].

[7]Notice that $\sigma_{[t_i,t_i]}(s)$ denotes the value of the event at t_i.

[8]In this paper, we assume $\forall t_i\in\mathcal{T}(s),\forall t_j\in\mathcal{T}(s), t_i\le t_j \Leftrightarrow i\le j$.

$$\begin{aligned}\sigma(s_1) &= \iota_1\ \iota_2\ \iota_1\ \iota_2\ \iota_3\ \iota_1\ \iota_2\ \tau\ \tau\ \tau\ \ldots\\ \sigma(s_2) &= \iota_1\ \iota_2\ \tau\ \tau\ \iota_1\ \tau\ \iota_2\ \iota_3\ \tau\ \iota_1\ \tau\ \iota_2\ \tau\ldots\\ \sigma(s_3) &= \iota_1\ \iota_2\ \tau\ \iota_1\ \iota_2\ \iota_3\ \tau\ \iota_1\ \iota_2\ \tau\ \tau\ \tau\ \ldots\end{aligned}$$

Fig. 2. Sequences of values of three latency equivalent signals.

Definition III.7: Two signals s_1, s_2 are latency equivalent $s_1 \equiv_\tau s_2$ iff $\mathcal{F}_\iota[\sigma(s_1)] = \mathcal{F}_\iota[\sigma(s_2)]$.

The *reference signal* s_{ref} of a class of latency equivalent signals is a strict signal obtained by assigning the sequence of informative values that characterizes the equivalence class to the first $|\mathcal{F}_\iota[\sigma(s_1)]|$ timestamps.

For instance, Fig. 2 reports the sequences associated to three signals that belong to the same latency equivalent class: signal s_1 is also the reference signal of the class.

Latency equivalent signals contain the same sequences of informative values, but with different timestamps. Hence, it is useful to identify their informative events with respect to the common reference signal: the *ordinal* of an informative event coincides with its position in the reference signal.

Definition III.8: The ordinal of an informative event $e_k = (v_k, t_k) \in \mathcal{E}_\iota(s)$ is defined as $\text{ord}(e_k) = |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}](s)| - 1$. Let s_1 and s_2 be two latency equivalent signals: two informative events $e_k(s_1) \in \mathcal{E}_\iota(s_1)$ and $e_l(s_2) \in \mathcal{E}_\iota(s_2)$ are said to be corresponding events iff $\text{ord}(e_k(s_1)) = \text{ord}(e_l(s_2))$. The slack between two corresponding events is defined as $\text{slack}(e_k(s_1), e_l(s_2)) = |k - l|$. Hence, if s is strict, the ordinal of an informative event coincides with its position on $\sigma(s)$. Observe that if s_1 and s_2 are latency equivalent signals, then corresponding informative events in s_1 and s_2 have the same ordinals (while they may have different timestamps).

We extend the notion of latency equivalence to behaviors in a componentwise manner.

Definition III.9: Two behaviors $(s_1, \ldots, s_N)$ and $(s'_1, \ldots, s'_N)$ are latency equivalent iff $\forall i\ (s_i \equiv_\tau s'_i)$. A behavior $b = (s_1, \ldots, s_N)$ is strict iff every signal $s_i \in b$ is strict. Every class of latency equivalent behaviors contains only one strict behavior. This is called the *reference behavior*.

Definition III.10: Two processes P_1 and P_2 are latency equivalent $P_1 \equiv_\tau P_2$ if every behavior of one is latency equivalent to some behavior of the other. A process P is strict iff every behavior $b \in P$ is strict. Every class of latency equivalent processes contains only one strict process: the *reference process*.

Definition III.11: A signal s_1 is latency dominated by another signal $s_2, s_1 \leq_\tau s_2$ iff $s_1 \equiv_\tau s_2$ and $T_1 \leq T_2$ with $T_k = \max\{t \,|\, t \in \mathcal{T}_\iota(s_k)\}, k = 1, 2$.

Hence, referring to the example of Fig. 2, signal s_3 is dominated by signal s_2 since $T_3 = 9$ while $T_2 = 12$. Notice that a reference signal is latency dominated by every signal belonging to its equivalence class. Latency dominance is extended to behaviors and processes as in the case of latency equivalence.

D. Ordering the Set of Informative Events

A total order among events of a behavior is necessary to develop our theory. In particular, we introduce an ordering among events that is motivated by causality: events that have a smaller ordinal are ordered before the ones with larger ordinals.[9] In addition, to avoid combinational cycles that may be created by processing events with the same ordinal, we rely on a well-founded order over the set of signals. This order in real-life designs corresponds to input–output combinational dependencies as they can be found, for instance, in the implementation of Mealy's finite-state machines. We cast this consideration in the most general form possible to extend maximally the applicability of our method by giving the following definition.

Definition III.12: Given a behavior $b = (s_1, \ldots, s_N)$, symbol $\leq_c$ denotes a well-founded order on its set of signals. The well-founded order induces a *lexicographic order* $\leq_{\text{lo}}$ over the set of informative events of b, s.t. for all pairs of events (e_1, e_2) with $e_1 \in \mathcal{E}_\iota(s_i)$ and $e_2 \in \mathcal{E}_\iota(s_j)$

$$\begin{aligned}e_1 \leq_{\text{lo}} e_2 \Leftrightarrow{} & [(\text{ord}(e_1) < \text{ord}(e_2))\\ & \vee ((\text{ord}(e_1) = \text{ord}(e_2)) \wedge (s_i \leq_c s_j))]. \quad (1)\end{aligned}$$

The following function returns the first informative event (in signal s_j of behavior b) following an event $e \in b$ with respect to the lexicographic order $\leq_{\text{lo}}$.

Definition III.13: Given a behavior $b = (s_1, \ldots, s_N)$ and an informative event $e(s_i) \in \mathcal{E}_\iota(s_i)$, the function *nextEvent* is defined as

$$\text{nextEvent}(s_j, e(s_i)) = \min_{e_k(s_j) \in \mathcal{E}_\iota(s_j)} \{e(s_i) \leq_{\text{lo}} e_k(s_j)\}.$$

A *stall move* postpones an informative event of a signal of a given behavior by one timestamp. The stall move is used to account for long delays along communication channels (i.e., wires on the chip) and to add delays where needed to guarantee functional correctness of the design.

Definition III.14: Given an informative event $e_k(s_j) = (v_k, t_k)$ in a behavior $b = (s_1, \ldots, s_j, \ldots, s_N)$, a stall move returns a behavior $b' = \text{stall}(e_k(s_j)) = (s_1, \ldots, s'_j, \ldots, s_N)$, s.t. for all $l \in \mathbb{N}$

$$\begin{gathered}\sigma_{[t_0, t_{k-1}]}(s'_j) = \sigma_{[t_0, t_{k-1}]}(s_j)\\ \sigma_{[t_k, t_k]}(s'_j) = \tau\\ \sigma_{[t_{k+l+1}, t_{k+l+1}]}(s'_j) = \sigma_{[t_{k+l}, t_{k+l}]}(s_j).\end{gathered}$$

A *procrastination effect* represents the "effect" of a stall move $\text{stall}(e_k(s_j))$ on other signals of behavior b in correspondence of events following $e_k(s_j)$ in the lexicographic order. The processes will "respond" to the insertion of stalls in some of their signals by "delaying" other signals that are causally related to the stalled signals. Given a behavior b for each stall move on events of b, we have a corresponding set of behaviors (the procrastination effect set).

Definition III.15: A procrastination effect is a point-to-set map that takes a behavior $b' = (s'_1, \ldots, s'_N) = \text{stall}(e_k(s_j))$ resulting from the application of a stall move on event $e_k(s_j)$ of behavior $b = (s_1, \ldots, s_N)$ and returns a set of behaviors $\mathcal{PE}[\text{stall}(e_k(s_j))]$ s.t. $b'' = (s''_1, \ldots, s''_N) \in \mathcal{PE}[b']$ iff the following three conditions hold:

1) $s''_j = s'_j$;

[9] Think of a strict process where the ordinal is related to the timestamp; the order implies that past events do not depend on future events.

$$b = \begin{cases} \sigma(s_1) &= \iota_1\ \tau\ \iota_2\ \tau\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_2\ \tau\ \iota_4\ \tau\ \iota_3\ \tau\ \iota_1\ \tau\ \iota_2\ \tau\ \tau\ \tau\ \ldots \\ \sigma(s_2) &= \tau\ \iota_5\ \tau\ \iota_1\ \tau\ \iota_6\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_5\ \tau\ \iota_6\ \tau\ \iota_7\ \tau\ \iota_8\ \tau\ \tau\ \tau\ \iota_7\ \tau\ \ldots \end{cases}$$

$$\textbf{Stall} \qquad \Downarrow \qquad \textbf{Move}$$

$$b' = \begin{cases} \sigma(s'_1) &= \iota_1\ \tau\ \iota_2\ \tau\ \boxed{\tau}\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_2\ \tau\ \iota_4\ \tau\ \iota_3\ \tau\ \iota_1\ \tau\ \iota_2\ \tau\ \tau\ \tau\ \ldots \\ \sigma(s'_2) &= \tau\ \iota_5\ \tau\ \iota_1\ \tau\ \iota_6\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_5\ \tau\ \iota_6\ \tau\ \iota_7\ \tau\ \iota_8\ \tau\ \tau\ \tau\ \iota_7\ \tau\ \ldots \end{cases}$$

Fig. 3. Behavior $b' = \text{stall}(e_5(s_1))$ is obtained stalling the fifth event of signal s_1 of behavior b.

2) $\forall i \in [1, N], i \neq j, s''_i \equiv_\tau s'_i$ and $\sigma_{[t_0, t_{l-1}]}(s''_i) = \sigma_{[t_0, t_{l-1}]}(s'_i)$, where t_l is the timestamp of event $e_l(s_i) = \text{nextEvent}(s_i, e_k(s_j))$;
3) $\exists K$ finite s.t. $\forall i \in [1, N], i \neq j, \exists k_i \leq K$, $\sigma_{[t_{l+k_i}, \infty]}(s''_i) = \sigma_{[t_l, \infty]}(s'_i)$.

Each behavior in $\mathcal{PE}[b']$ is obtained from b' by possibly inserting other stalling events in any signal of b', but only at "later" timestamps, i.e., to postpone informative event that follows $e_k(s_j)$ with respect to the lexicographic order $\leq_{\text{lo}}$. Observe that a procrastination effect returns a behavior that latency dominates the original behavior.

E. Compositionality of Patient Processes

We are now ready to define the notion of patient process, which lies at the core of the theory. A patient process can take stall moves on any signal of its behaviors by reacting with the appropriate procrastination effects. Patience is the key condition for the intellectual property (IP) blocks to be combinable according to our method. The following theorems guarantee that the notion of latency equivalence of processes for patient processes is compositional .

Definition III.16: A process P is patient iff

$$\begin{aligned} &\forall b = (s_1, \ldots, s_N) \in P, \quad \forall j \in [1, N] \\ &\forall e_k(s_j) \in \mathcal{E}_\iota(s_j) \\ &(\mathcal{PE}[\text{stall}(e_k(s_j))] \cap P \neq \emptyset). \end{aligned}$$

Hence, the result of a stall move on one of the events of a patient process may not satisfy the process, but one of the behaviors of the procrastination effect corresponding to the stall move does satisfy the process, i.e., if we stall a signal on an input to a functional block, the block will be forced to delay some of its outputs or if we request an output signal to be delayed then an appropriate delay may be added to some of its incoming inputs.

Lemma III.1: Let P_1 and P_2 be two patient processes. Let $b_1 \in P_1, b_2 \in P_2$ be two behaviors with the same lexicographic order s.t. $b_1 \equiv_\tau b_2$. Then, there exists a behavior $b' \in (P_1 \cap P_2), b_1 \equiv_\tau b' \equiv_\tau b_2$.

Proof (Constructive): Let $b_1 = (r_1, \ldots, r_N) \in P_1$ and $b_2 = (q_1, \ldots, q_N) \in P_2$ be the two behaviors with the same lexicographic order. Since b_1 and b_2 are latency equivalent, each event in b_1 has a corresponding event in b_2 and vice versa (see Definition III.8). Let $r_1 \equiv_\tau q_1, \ldots, r_N \equiv_\tau q_N$. Let $W = \{w \mid \exists k \in \mathbb{N}, \exists l \in \mathbb{N}(k \neq l \wedge \text{ord}(e_k(r_j)) = \text{ord}(e_l(q_j)) = w)\}$ be the set of ordinals associated to pairs of corresponding events of b_1 and b_2 whose timestamps differ. Define the distance between behaviors b_1, b_2 as

$$d(b_1, b_2) = \begin{cases} \max\left\{\frac{1}{2^w} \mid w \in W\right\}, & \text{if } W \neq \emptyset \\ 0, & \text{otherwise} \end{cases}.$$

This distance is reminiscent of the Cantor metric. Thus, b_1 and b_2 have distance equal to zero if all pairs of corresponding events are *aligned*.[10] In this case, b_1 and b_2 are identical, i.e., they are the same behavior that belongs to $(P_1 \cap P_2)$. Now, suppose that $d(b_1, b_2) = (1/2^{w_1}) \neq 0$. In this case, w_1 is the smallest ordinal among those which are associated to unaligned pairs of corresponding events. Without loss of generality, let $p_1 = (e_k(r_j), e_l(q_j))$ be the pair of corresponding events whose ordinal is equal to w_1 and let $l > k$. Apply a stall move to $e_k(r_j)$ to obtain a new behavior $b'_1 = (s'_1, \ldots, s'_N) = \text{stall}(e_k(r_j)) \equiv_\tau b_1$. Obviously, $\text{slack}(e_{k+1}(s'_j), e_l(q_j)) = \text{slack}(e_k(r_j), e_l(q_j)) - 1$. Note that b'_1 is not necessarily a behavior of P_1. However, since P_1 is patient, there exists $b''_1 = (s''_1, \ldots, s''_N) \equiv_\tau b_1$ s.t. $b'' \in \mathcal{PE}(\text{stall}(e_k(r_j)) \cap P_1$. Since, by definition of procrastination effect, $s''_j = s'_j$, then also $\text{slack}(e_{k+1}(s''_j), e_l(q_j)) = \text{slack}(e_k(r_j), e_l(q_j)) - 1$. Since the procrastination effect may postpone only events following $e_k(r_j)$ in the lexicographic order $\leq_{\text{lo}}$, then all the pairs of corresponding events of b''_1 and b_2 with ordinal smaller than w are still aligned. Now, there are two possibilities: if $\text{slack}(e_{k+1}(s''_j), e_l(q_j)) = 0$, then one more pair has been aligned and $d(b''_1, b_2) < d(b_1, b_2)$; otherwise, we can reduce by one this slack by repeating the same procedure starting from behavior b''_1. In any case, after $l - k$ steps of the procedure outlined above, we obtain a behavior $b^\star_1 \equiv_\tau b_1$ that satisfies P_1 and s.t. $d(b^\star_1, b_2) < d(b_1, b_2)$ because one more pair of corresponding events has been aligned. We say that we have performed an *alignment step*.

Now, if $d(b^\star_1, b_2) = 0$, then there are no more unaligned pairs, the two behaviors are identical, and the lemma is proven since $b^\star_1 \equiv_\tau b_1$. Instead, if $d(b^\star_1, b_2) = (1/2^{w_2}) \neq 0$, then we consider the next unaligned pair p_2 of corresponding events and we execute a second alignment step. Note that at the mth step, while aligning pair p_m with ordinal w_m, we may increase the slack of some of the pairs following p_m in the lexicographic order, but we keep aligned all the pairs preceding p_m. During this sequence of alignment steps, we obtain two sequences of behaviors (one of behaviors in P_1 latency equivalent to b_1 and one of behaviors in P_2 latency equivalent to b_2) whose distance is decreasing monotonically. Since both b_1 and b_2 contain the

[10]A pair of corresponding events is said *aligned* if the events are synchronous or, according to Definition III.8, if their slack is zero.

$$\begin{tikzcd}
& b^\star = \iota_1\ \tau\ \ \iota_2\ \tau\ \ \tau\ \ \iota_3\ \tau\ \ \iota_4\ \tau\ \ \tau\ \ \iota_5\ \tau\ \ \tau\ \ \dots & \\
b_1 = \iota_1\ \tau\ \ \iota_2\ \ \iota_3\ \tau\ \ \iota_4\ \tau\ \ \tau\ \ \iota_5\ \tau\ \ \tau\ \ \dots \arrow[ur] & & b_2 = \iota_1\ \ \iota_2\ \tau\ \ \tau\ \ \iota_3\ \ \iota_4\ \tau\ \ \iota_5\ \tau\ \ \tau\ \ \dots \arrow[ul] \\
& b_{ref} = \iota_1\ \ \iota_2\ \ \iota_3\ \ \iota_4\ \ \iota_5\ \tau\ \ \tau\ \ \dots \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Fig. 4. Sketch for proof on compositionality of latency equivalence.

same finite number of informative events,[11] the set U of pairs of unaligned corresponding events is also finite. The slack of each of these pair is also a finite number. At the mth step, we have at most h_m substeps to align p_m, where h_m is the starting slack for p_m. In the worst case, each behavior $b^\star$ obtained during the substeps of the alignment step may have slacks of all the remaining unaligned pairs increased by at most K (see Definition III.15). Hence, at the end of the mth step, $|U|$ has been decreased by one, while all the slacks of its remaining elements have been increased by at most $h_m \cdot K$, a finite number. Thus, for $|U| \geq l > m$, the new slacks for the remaining unaligned pairs is $h'_l \leq h_l + h_m \cdot K$. Globally, we perform in the worst case $|U|$ alignment steps and for each of them we have a finite number of substeps. Hence, the two sequences of behaviors are also finite and the last elements of these sequences do not have unaligned pairs and, therefore, have distance equal to zero. □

Theorem III.1: If P_1 and P_2 are patient processes, then $(P_1 \cap P_2)$ is a patient process.

Proof: Let $b = (s_1, \dots, s_N)$ be a behavior in $P_1 \cap P_2$. Consider behaviors $b_1 = (r_1, \dots, r_N) \in P_1$ and $b_2 = (q_1, \dots, q_N) \in P_2$, s.t. $b_1 = b_2 = b$. For all $j \in [1, N]$ and for all $k \in \mathbb{N}$, let $e_k(s_j) \in \mathcal{E}_\iota(s_j)$. Since $b_1 = b_2 = b$, then $e_k(r_j) \in \mathcal{E}_\iota(r_j)$ and $e_k(q_j) \in \mathcal{E}_\iota(q_j)$. Let $b' = \mathrm{stall}(e_k(s_j)) \equiv_\tau b$. Similarly, $b'_1 = \mathrm{stall}(e_k(r_j)) \equiv_\tau b_1$ and $b'_2 = \mathrm{stall}(e_k(s_j)) \equiv_\tau b_2$. Since P_1 is patient there exists a behavior $b''_1 \equiv_\tau b_1$ s.t. $b''_1 \in \mathcal{PE}[b'_1] \cap P_1$ and since P_2 is patient, there exists a behavior $b''_2 \equiv_\tau b_2$ s.t. $b''_2 \in \mathcal{PE}[b'_2] \cap P_2$. Notice that $b_1 = b_2$ implies that $b''_1 \equiv_\tau b''_2$. However, it is not necessarily the case that $b''_1 = b''_2$. In fact, procrastination effects may have misaligned pairs of corresponding informative signals that come after $(e_k(r_j), e_k(s_j))$ with respect to lexicographic order $\leq_{\mathrm{lo}}$. Since $b_1 = b_2$ share the same lexicographic order, by Lemma III.1, there exists a behavior $b'' \equiv_\tau b''_1 \equiv_\tau b''_2$ s.t. $b'' \in P_1 \cap P_2$. The construction of b'' given in the proof of Lemma III.1 involves only unaligned pairs of corresponding events between b''_1 and b''_2 and all these unaligned pairs correspond to informative events that come after $e_k(s_j)$ with respect to lexicographic order $\leq_{\mathrm{lo}}$. Further, since the number of informative events is finite, the number of unaligned pairs is also finite. Hence, each signal s''_i of b'' is obtained by inserting a finite number of stalling events not earlier than timestamp t_l with $e_l(s_i) = \mathrm{nextEvent}(s_i, e_k(s_j))$. Therefore, by Definition III.15, $b'' \in \mathcal{PE}(\mathrm{stall}(e_k(s_j)))$. Since we have already seen that $b'' \in P_1 \cap P_2$, then $(P_1 \cap P_2)$ is a patient process. □

Fig. 4 illustrates the above proof for the case when the two behaviors are just 1-tuple signals.

Theorem III.2: For all patient processes P_1, P_2, P'_1, P'_2, if $P_1 \equiv_\tau P'_1$ and $P_2 \equiv_\tau P'_2$ then $(P_1 \cap P_2) \equiv_\tau (P'_1 \cap P'_2)$.

Proof: Let $b = (s_1, \dots, s_N)$ be a behavior in $P_1 \cap P_2$. Latency equivalence implies that there must be behaviors $b_1 = (r_1, \dots, r_N) \in P'_1$ and $b_2 = (q_1, \dots, q_N) \in P'_2$ such that $b_1 \equiv_\tau b \equiv_\tau b_2$. Since b_1 and b_2 are latency equivalent and P'_1 and P'_2 are patient, Lemma III.1 guarantees that there must be a latency equivalent behavior $b' \in (P'_1 \cap P'_2)$. The other direction of the proof is symmetric. □

Therefore, we can replace any process in a system of patient processes by a latency equivalent process, and the resulting system will be latency equivalent. A similar theorem holds for replacing strict processes with patient processes.

Theorem III.3: For all strict processes P_1, P_2 and all patient processes P'_1, P'_2, if $P_1 \equiv_\tau P'_1$ and $P_2 \equiv_\tau P'_2$, then $(P_1 \cap P_2) \equiv_\tau (P'_1 \cap P'_2)$.

Proof: The argument that every behavior in $(P_1 \cap P_2)$ has an equivalent in $(P'_1 \cap P'_2)$ is as in Theorem III.2. For the other direction, let b' be a behavior in $P'_1 \cap P'_2$. Latency equivalence implies that there must be behaviors $b_1 \in P_1$ and $b_2 \in P_2$ such that $b_1 \equiv_\tau b' \equiv_\tau b_2$. Since P_1 and P_2 are strict, b_1 and b_2 are also strict. Being latency equivalent, they must, therefore, be equal. Thus, $b_1 \in (P_1 \cap P_2)$. □

This means that we can replace all processes in a system of strict processes by corresponding patient processes and the resulting system will be latency equivalent to the original one. This is the core of our idea: take a design based on the assumption that computation in one functional module and communication among modules "take no time" (synchronous hypothesis),[12] i.e., the processes corresponding to the functional modules and their composition are strict and replace it with a design where communication does take time (more than one clock cycle) and, as a result, signals are delayed, but without changing the sequence of informative events observed at the system level, i.e., with a set of patient processes.

IV. Patient Processes and Relay Stations

As explained in Section I, one of the goals of the latency-insensitive design methodology is to be able to "pipeline" a communication channel by inserting an arbitrary amount of memory elements. In the framework of our theory, this operation corresponds to adding some particular processes, called *relay stations*, to the given system. In this section, we give the formal definition of a relay station and we show how patient systems (i.e., systems of patient processes) are insensitive to the insertion of relay stations. In Section IV-A, we discuss under which

[11]Recall that the number of informative events for every behavior considered in latency-insensitive designs is finite.

[12]In other words, communication and computation are completed in one clock cycle.

assumption a generic system can be transformed into a patient system.

A. Channels and Buffers

The tagged signal model provides the notion of channel to formalize the composition of processes [33]. A *channel* is a connection[13] constraining two signals to be identical.

Definition IV.1: A *channel* $C(i,j) \subset \mathcal{S}^N, i,j \in [1,N]$ is a process s.t.

$$b = (s_1, \ldots, s_N) \in C(i,j) \Leftrightarrow s_i = s_j.$$

As the following lemma formally proves, a channel is not a patient process because it lacks the capacity of storing an event and delaying its communication between two processes.

Lemma IV.1: A channel $C(i,j) \subset \mathcal{S}^N$ is not a patient process.

Proof: Let $b = (s_1, \ldots, s_N)$ be a behavior of a channel $C(i,j)$ and, without loss of generality, suppose that $s_i \leq_c s_j$. Consider a pair of corresponding informative events in s_i and s_j : $e_k(s_i) = (v_1, t_k)$ and $e_k(s_j) = (v_2, t_k)$. Since $b \in C(i,j)$, then $s_i = s_j$ and, therefore, $v_1 = v_2 \neq \tau$. Moreover, $s_i = s_j$ implies that $\mathrm{ord}(e_k(s_i)) = \mathrm{ord}(e_k(s_j))$, since $s_i \leq_c s_j, e_k(s_j) = \mathrm{nextEvent}(s_j, e_k(s_i))$. Without loss of generality, suppose that $e_k(s_i)$ and $e_k(s_j)$ are followed by $(l-1) > 0$ stalling events, i.e., formally, $l > 1, |\mathcal{F}_\iota[\sigma_{[t_k, t_{k+(l-1)}]}(s_i)]| = |\mathcal{F}_\iota[\sigma_{[t_k, t_{k+(l-1)}]}(s_j)]| = 0$. Then, consider informative event $e_{k+l}(s_i) = (v_{k+l}, t_{k+l})$. By Definition III.13, $e_{k+l}(s_i) = \mathrm{nextEvent}(s_i, e_k(s_j))$. Now, let $b' = (s'_1, \ldots, s'_N) = \mathrm{stall}(e_k(s_j))$ be the behavior obtained by applying a stall move on $e_k(s_j)$. At timestamp t_k, s'_j presents a stalling event, while the event of s'_j corresponding to $e_k(s_j)$ is $e_{k+1}(s'_j) = (v_2, t_{k+1})$, which occurs at timestamp t_{k+1}. Then, consider any behavior $b'' = (s''_1, \ldots, s''_N) \in \mathcal{PE}[b']$. By Definition III.15, since $e_{k+l}(s_i) = \mathrm{nextEvent}(s_i, e_k(s_j))$, then $\sigma_{[t_0,t_k]}(s''_i) = \sigma_{[t_0,t_k]}(s'_i) = \sigma_{[t_0,t_k]}(s_i)$. In particular, $\sigma_{[t_k,t_k]}(s''_i) = \sigma_{[t_k,t_k]}(s_i) \neq \tau$ and, therefore, $\sigma_{[t_k,t_k]}(s''_i) \neq \sigma_{[t_k,t_k]}(s''_j)$, which, finally, implies that $s''_i \neq s''_j$. Hence, $\forall b'' \in \mathcal{PE}[b'](b'' \notin C(i,j))$ and, by Definition III.16, $C(i,j)$ is not patient. □

Hence, to formally model communication delays as well as pipeline stages, we introduce the notion of buffer: a buffer is a process relating two signals s_i, s_j of a behavior b and is defined by means of three parameters: capacity c, minimum forward latency l_f, and minimum backward latency l_b. A buffer forces signals s_i, s_j to be latency equivalent and to satisfy the following relationships for all natural numbers k.

1) The difference between the amount of information events seen at s_i from timestamp zero to timestamp $k - l_f$ and the amount of informative events seen at s_j from timestamp zero to timestamp k is greater or equal than zero.
2) The difference between the amount of information events seen at s_i from timestamp zero to timestamp k and the amount of informative events seen at s_j from timestamp zero to timestamp $k - l_b$ is at most c.

[13]See Section III-A for the definition of connection.

Definition IV.2: A *buffer* $B^c_{l_f,l_b}(i,j)$ with capacity $c \geq 0$, minimum forward latency $l_f \geq 0$, and minimum backward latency $l_b \geq 0$ is a process s.t. $\forall i,j \in [1,N]$: $b = (s_1, \ldots, s_N) \in B^c_{l_f,l_b}(i,j)$ iff $(s_i \equiv_\tau s_j)$ and $\forall k \in \mathbb{N}$

$$0 \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-l_f)}]}(s_i)\right]\right| - \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_j)\right]\right| \tag{2}$$

$$c \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_i)\right]\right| - \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-l_b)}]}(s_j)\right]\right|. \tag{3}$$

By definition, given a pair of indexes $i,j \in [1,N]$, for all $l_b, l_f, c \geq 0$, all buffers $B^c_{l_f,l_b}(i,j)$ are latency equivalent. Observe also that buffer $B^0_{0,0}(i,j)$ coincides with channel $C(i,j)$ and, therefore, is not a patient process. In particular, we are interested in buffers having unitary latencies and we want to establish under which conditions such buffers are patient processes.

Lemma IV.2: If s_i, s_j are two signals s.t. $s_i \equiv_\tau s_j$ and $s_i \leq_c s_j$, then

1) $\forall g \in \mathbb{N}$ s.t. $e_g(s_i) \in \mathcal{E}_\iota(s_i)$, $\mathrm{nextEvent}(s_j, e_g(s_i))$ is the corresponding event of $e_g(s_i)$ in s_j;
2) $\forall h \in \mathbb{N}$ s.t. $e_h(s_j) \in \mathcal{E}_\iota(s_j), \mathrm{nextEvent}(s_i, e_h(s_j)) = \mathrm{nextEvent}(s_i, e_g(s_i))$, where $e_g(s_i)$ is the corresponding event of $e_h(s_j)$ in s_i.

Proof: Let $e_h(s_j)$ be the corresponding event of $e_g(s_i)$. By Definition III.8, $\mathrm{ord}(e_g(s_i)) = \mathrm{ord}(e_h(s_j) = k$. By Definition III.12, since $s_i \leq_c s_j$, then $e_g(s_i) \leq_{\mathrm{lo}} e_h(s_j)$. For all informative events, $e'(s_j)$ with $\mathrm{ord}(e'(s_j)) < k$ is clearly $e_g(s_i) \not\leq_{\mathrm{lo}} e'(s_j)$. Instead, for all informative events $e''(s_j)$ with $\mathrm{ord}(e''(s_j)) > k$, we have $e_g(s_i) \leq_{\mathrm{lo}} e''(s_j)$. However, $e_h(s_j)$ is clearly the minimum ordinal informative event of s_j that follows $e_g(s_i)$ with respect to the lexicographic order $\leq_{\mathrm{lo}}$ and, therefore, by Definition III.13, $\mathrm{nextEvent}(s_j, e_g(s_i)) = e_h(s_j)$. The second relation can be easily proven using the previous relation. We know that $\mathrm{nextEvent}(s_j, e_g(s_i)) = e_h(s_j)$, where $e_g(s_i)$ and $e_h(s_j)$ are corresponding events. Let $e'_g(s_i)$ be $\mathrm{nextEvent}(s_i, e_g(s_i))$. Then, necessarily, $\mathrm{ord}(e'_g(s_i)) = \mathrm{ord}(e_g(s_i)) + 1 = k + 1$. Since $\mathrm{ord}(e_h(s_j)) = \mathrm{ord}(e_g(s_i)) = k$, then, by Definition III.12, $e_h(s_j) \leq_{\mathrm{lo}} e'_g(s_i)$. Furthermore, $e'_g(s_i)$ is also the minimum ordinal event of s_i, which comes after $e_h(s_j)$ according to lexicographic order $\leq_{\mathrm{lo}}$ and, therefore, by Definition III.13, $\mathrm{nextEvent}(s_i, e_h(s_j)) = e'_g(s_i) = \mathrm{nextEvent}(s_i, e_g(s_i))$. □

Fig. 5 illustrates the previous lemma.

Theorem IV.1: Let $l_b = l_f = 1$. For all $c \geq 1$, $B^c_{1,1}(i,j)$ is patient iff $s_i \leq_c s_j$.

Proof: First, if $l_b = l_f = 1$, then inequalities (2) and (3) become

$$0 \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-1)}]}(s_i)\right]\right| - \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_j)\right]\right| \tag{4}$$

$$c \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_i)\right]\right| - \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-1)}]}(s_j)\right]\right|. \tag{5}$$

Only If: By contradiction, we prove that if $s_i \not\leq_c s_j$, then $B^c_{1,1}(i,j)$ is not a patient process. Suppose $s_i \not\leq_c s_j$. For all $c \geq 1$, let $b = (s_1, \ldots, s_N)$ be a behavior of $B^c_{1,1}(i,j)$ s.t. $\sigma(s_i) = \iota_0\ \iota_1\ \tau\ \tau\ \tau \ldots$ and $\sigma(s_j) = \tau\ \iota_0\ \iota_1\ \tau\ \tau\ \tau\ \ldots$. Let $b' = (s'_1, \ldots, s'_N) = \mathrm{stall}(e_0(s_i))$ with $e_0(s_i) = (\iota_0, t_0)$.

$$\sigma(s_i) = \underbrace{\boxed{k-1 \text{ informative events and } (g-k-1)\ \tau \text{ events}}}_{\cdot \leq_{lo} e_h(s_j)}\ \underbrace{\boxed{e_g}\ \boxed{\tau \text{ events}}\ \boxed{e_{g+1}}\ \cdots}_{e_h(s_j) \leq_{lo} \cdot}$$

$$\sigma(s_j) = \underbrace{\boxed{k-1 \text{ informative events and } (h-k-1)\ \tau \text{ events, } h \geq g}}_{\cdot \leq_{lo} e_g(s_i)}\ \underbrace{\boxed{e_h}\ \boxed{\tau \text{ events}}\ \boxed{e_{h+1}}\ \cdots}_{e_g(s_i) \leq_{lo} \cdot}$$

Fig. 5. Sketch to illustrate Lemma IV.2.

Clearly, $b' \notin B^c_{1,1}(i,j)$ because inequality (4) does not hold for $k=1$ since $s'_j = s_j$. Further, for all $b'' = (s''_1, \ldots, s''_N) \in \mathcal{PE}(\text{stall}(e_0(s_i)))$, we can prove that $b'' \notin B^c_{1,1}(i,j)$. In fact, since $s''_i = s'_i, b'' \in B^c_{1,1}(i,j)$ iff $\sigma_{[t_1,t_1]}(s''_j) = \tau$. However, consider that $\text{ord}(e_0(s_i)) = \text{ord}(e_1(s_j))$ and, since $s_i \not\leq_c s_j$, then $e_0(s_i) \not\leq_{\text{lo}} e_1(s_j)$. Further, $\text{ord}(e_0(s_i)) = \text{ord}(e_2(s_j)) - 1$. Therefore, $e_2(s_j) = \text{nextEvent}(s_j, e_0(s_i))$. Recall that by Definition III.15 of procrastination effect, $\sigma_{[t_0,t_{l-1}]}(s''_j) = \sigma_{[t_0,t_{l-1}]}(s'_j)$, where t_l is the timestamp of event $\text{nextEvent}(s_j, e_0(s_i))$. Hence, in our case, $t_l = t_2$ and $\sigma_{[t_0,t_1]}(s''_j) = \sigma_{[t_0,t_1]}(s'_j)$. Since $s'_j = s_j$, $\sigma_{[t_1,t_1]}(s''_j) = \sigma_{[t_1,t_1]}(s'_j) = \sigma_{[t_1,t_1]}(s_j) = \iota_0 \neq \tau$. This implies that $b'' \notin B^c_{1,1}(i,j)$. Hence, $\mathcal{PE}[\text{stall}(e_0(s_i))] \cap B^c_{1,1}(i,j) = \emptyset$ and $B^c_{1,1}(i,j)$ is not patient.

If: We prove that if $s_i \leq_c s_j$, then $B^c_{1,1}(i,j)$ is patient. For all $c \geq 1$, let $b = (s_1, \ldots, s_N)$ be a behavior of $B^c_{1,1}(i,j)$. We must analyze three distinct cases in which we, respectively, stall an informative signal of s_i, s_j and s_n with $n \in ([1,N]/\{i,j\})$.

1) For all $g \in \mathbb{N}$, such that $e_g(s_i) \in \mathcal{E}_\iota(s_i)$, let $b' = (s'_1, \ldots, s'_N) = \text{stall}(e_g(s_i)) \equiv_\tau b$. Since $s'_j = s_j, b' \notin B^c_{1,1}(i,j)$ iff inequality (4) does not hold for some $k \in \mathbb{N}$. In fact, b' satisfies the other two conditions of Definition IV.2 because $b' \equiv_\tau b$ and to insert a stalling event on s_i (while s_j remains the same) cannot induce a violation of inequality (5). Now, suppose first that b' satisfies also inequality (4) for all $k \in \mathbb{N}$: then, there exists at least a behavior that belongs to $\mathcal{PE}[\text{stall}(e_g(s_i))] \cap B^c_{1,1}(i,j)$ and this behavior is b' because $\forall g \forall i, \text{stall}(e_g(s_i)) \in \mathcal{PE}[\text{stall}(e_g(s_i))]$. A more interesting case is when inequality (4) does not hold: in this case, $b' \notin B^c_{1,1}(i,j)$. Then, consider a behavior $b'' = (s''_1, \ldots, s''_N)$ s.t. $\forall n \in [1,N], n \neq j, (s''_n = s'_n)$, while s''_j is obtained from s'_j by inserting a stalling event at timestamp t_h, where t_h is also the timestamp of event $e_h(s_j) = \text{nextEvent}(s_j, e_g(s_i))$. Clearly, this construction guarantees that $b'' \in \mathcal{PE}[\text{stall}(e_g(s_i))]$. It remains to be proven that $b'' \in B^c_{1,1}(i,j)$. First, by construction, $b'' \equiv_\tau b$. Then, check whether s''_i, s''_j satisfy inequalities (4) and (5) for all $k \in \mathbb{N}$. First, since $s_i \equiv_\tau s_j$ and $s_i \leq_c s_j$, by Lemma IV.2, $e_h(s_j) = \text{nextEvent}(s_j, e_g(s_i))$ is the corresponding event of $e_g(s_i)$ in s_j. Hence, $\text{ord}(e_g(s_i)) = \text{ord}(e_h(s_j)) = \text{ord}(e_g(s''_i)) = \text{ord}(e_h(s''_j))$ and, recalling Definition III.8, $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_i)]| = |\mathcal{F}_\iota[\sigma_{[t_0,t_h]}(s_j)]|$. Since, by hypothesis, s_i, s_j satisfy inequality (4) for all $k \in \mathbb{N}$, then $g < h$. Compare s''_i and s''_j, respectively, with s_i and s_j: s''_i has been derived by s_i inserting a τ at t_g, while s''_j has been derived by s_j inserting a τ at t_h. Hence, we can derive the following four equations. Further, each term in these equations can be bounded using the fact that s_i, s_j satisfy inequalities (4) and (5) for all $k \in \mathbb{N}$

$$\forall k \in [0, g-1],$$
$$\left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s''_i)\right]\right| = \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_i)\right]\right| \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-1}]}(s_j)\right]\right| + c \tag{6}$$
$$\forall k \in [g, \infty[,$$
$$\left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s''_i)\right]\right| = \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-1}]}(s_i)\right]\right| \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-2}]}(s_j)\right]\right| + c \tag{7}$$
$$\forall k \in [0, h-1],$$
$$\left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s''_j)\right]\right| = \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_j)\right]\right| \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-1}]}(s_i)\right]\right| \tag{8}$$
$$\forall k \in [h, \infty[,$$
$$\left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s''_j)\right]\right| = \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-1}]}(s_j)\right]\right| \leq \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-2}]}(s_i)\right]\right| \tag{9}$$

Now, keeping in mind that $g < h$, it is easy to prove that:

a) using (8) and (6), s''_i, s''_j satisfy (4), $\forall k \in [0, g-1]$;
b) using[14] (8) and (7), s''_i, s''_j satisfy (4), $\forall k \in [g, h-1]$;
c) using (9) and (7), s''_i, s''_j satisfy (4), $\forall k \in [h, \infty[$;
d) using (6) and (8), s''_i, s''_j satisfy (5), $\forall k \in [0, g-1]$;
e) using (7) and (9), s''_i, s''_j satisfy (5), $\forall k \in [g, h-1[$;
f) using (7) and (9), s''_i, s''_j satisfy (5), $\forall k \in [h, \infty[$.

Therefore, $b'' \in B^c_{1,1}(i,j)$.

2) Consider now $b' = (s'_1, \ldots, s'_N) = \text{stall}(e_h(s_j)) \equiv_\tau b$, where for all $h \in \mathbb{N}, e_h(s_j) \in \mathcal{E}_\iota(s_j)$. Let $e_q(s_i) = \text{nextEvent}(s_i, e_h(s_j))$ and $e_p(s_i)$ be the corresponding event of $e_h(s_j)$ in s_i: then, since $s_i \equiv_\tau s_j$ and $s_i \leq_c s_j$, by Lemma IV.2, $e_q(s_i) = \text{nextEvent}(s_i, e_p(s_i))$. Now, construct $b'' = (s''_1, \ldots, s''_N)$ in such a way that $\forall n \in [1,N], n \neq i, (s''_n = s'_n)$, while s''_i is obtained from s'_i by inserting a stalling event at timestamp t_g, where $g = \min_{k \in [h+1,\infty[}\{k \mid e_k(s_i) \in \mathcal{E}_\iota(s_i)\}$. Hence, if $q > h$, then $e_g(s_i) = e_q(s_i)$, else $e_q(s_i) \leq_{\text{lo}} e_g(s_i)$. In both cases, this construction guarantees that $b'' \in \mathcal{PE}[\text{stall}(e_h(s_j))]$. It remains to be proven that $b'' \in B^c_{1,1}(i,j)$. First, by construction, $b'' \equiv_\tau b$. Then, check whether s''_i, s''_j satisfy inequalities (4) and (5) for all $k \in \mathbb{N}$. Compare s''_i and s''_j, respectively, with s_i and s_j: s''_i has been derived by s_i inserting a τ at

[14]Recall that $\sigma_{[t_k,t_k]}(s_i) = \tau$.

$$B^1_{1,1}\begin{cases}\sigma(s_1) &= \iota_1\ \tau\ \iota_2\ \tau\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_5\ \tau\ \iota_6\ \tau\ \iota_7\ \tau\ \iota_8\ \tau\ \iota_9\ \tau\ \tau\ \tau\ \iota_{10}\ \tau\ \ldots\\ \sigma(s_2) &= \tau\ \iota_1\ \tau\ \iota_2\ \tau\ \iota_3\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_5\ \tau\ \iota_6\ \tau\ \iota_7\ \tau\ \iota_8\ \tau\ \tau\ \tau\ \iota_9\ \tau\ \iota_{10}\ \tau\ \ldots\end{cases}$$

$$B^2_{1,1}\begin{cases}\sigma(s_1) &= \iota_1\ \iota_2\ \iota_3\ \tau\ \tau\ \iota_4\ \iota_5\ \iota_6\ \tau\ \tau\ \tau\ \iota_7\ \tau\ \iota_8\ \iota_9\ \iota_{10}\ \ldots\\ \sigma(s_2) &= \tau\ \iota_1\ \iota_2\ \iota_3\ \tau\ \tau\ \iota_4\ \tau\ \tau\ \tau\ \iota_5\ \iota_6\ \iota_7\ \tau\ \iota_8\ \iota_9\ \iota_{10}\ \ldots\end{cases}$$

Fig. 6. Comparing two possible behaviors of finite buffers $B^1_{1,1}(s_1, s_2)$ and $B^2_{1,1}(s_1, s_2)$.

t_g, while s''_j has been derived by s_j inserting a τ at t_h. Hence, previous relations (6)–(9) hold also in this case. Now, keeping in mind that here $h < g$, it is easy to prove that:

a) using (8) and (6), s''_i, s''_j satisfy (4), $\forall k \in [0, h-1]$;
b) using (9) and (6), s''_i, s''_j satisfy (4), $\forall k \in [h, g-1]$;
c) using (9) and (7), s''_i, s''_j satisfy (4), $\forall k \in [g, \infty[$;
d) using (6) and (8), s''_i, s''_j satisfy (5), $\forall k \in [0, h-1]$;
e) using[15] (6) and (8), s''_i, s''_j satisfy (5), $\forall k \in [h, g-1[$;
f) using (7) and (9), s''_i, s''_j satisfy (5), $\forall k \in [g, \infty[$.

Therefore, $b'' \in B^c_{1,1}(i,j)$ in this case, too.

3) Finally, for all $n \in ([1,N]/\{i,j\})$ let $b' = (s'_1, \ldots, s'_N) = \text{stall}(e_h(s_n)) \equiv_\tau b$, where for all $h \in \mathbb{N}, e_h(s_n) \in \mathcal{E}_\iota(s_n)$. Then, trivially, $b' \in \mathcal{PE}[\text{stall}(e_h(s_n))] \cap B^c_{1,1}(i,j)$.

In conclusion, combining all three cases, we have

$$\begin{aligned}&\forall b = (s_1, \ldots, s_N) \in B^c_{1,1}(i,j), \quad \forall n \in [1,N],\\ &\quad \forall e_k(s_n) \in \mathcal{E}_\iota(s_n),\\ &\qquad (\mathcal{PE}[\text{stall}(e_k(s_n))] \cap B^c_{1,1}(i,j) \neq \emptyset).\end{aligned}$$

Hence, $B^c_{1,1}(i,j)$ is patient. □

Consider a strict system $P_{\text{strict}} = \cap_{m=1}^{M} P_m$ with N strict signals $s_1, \ldots, s_N$. As explained in Section III-A, processes can be defined over different signal sets and to compose them we may need to formally extend the set of signals of each process to contain all the signals of all processes. However, without loss of generality, consider the particular case of composing M processes that are already defined on the same N signals. Hence, any generic behavior $b_m = (s_{m_1}, \ldots, s_{m_N})$ of P_m is also a behavior of P_{strict} iff for all $l \in [1, M], l \neq m$, process P_l contains a behavior $b_l = (s_{l_1}, \ldots, s_{l_N})$ s.t. $\forall n \in [1, N]$ $(s_{l_n} = s_{m_n})$. In fact, we may assume to derive system P_{strict} by connecting the M processes with $(M-1)\cdot N$ channel processes $C(l_n, (l+1)_n)$, where $l \in [1, (M-1)]$ and $n \in [1, N]$. Further, we may also assume to "decompose" any channel process $C(m_n, l_n)$ with an arbitrary number X of channel processes $C(m_n, x_1), C(x_1, x_2), \ldots, C(x_{X-1}, l_n)$ by adding $X-1$ auxiliary signals, each of them forced to be equal to $m_n = l_n$. The theory developed in Section III guarantees that if we replace each process $P_m \in P_{\text{strict}}$ with a latency equivalent patient process and each channel $C(i,j)$ with a patient buffer $B^1_{1,1}(i,j)$, we obtain a system P_{patient}, which is patient and latency equivalent to P_{strict}. In fact, *having a patient buffer in a patient system is equivalent to having a channel in a strict system*. Since "decomposing" a channel $C(i,j)$ has no observable effect on a strict system, we are, therefore, free to add an arbitrary number of patient buffers into the corresponding patient system to replace this channel. Since we use patient buffers with unitary latencies, we can distribute them along that long wire on the chip which implements $C(i,j)$ in such a way that the wire gets decomposed in segments whose physical lengths can be spanned in a single real clock cycle (as anticipated in Section I).

[15]Recall that $\sigma_{[t_h, t_h]}(s_j) = \tau$.

B. Relay Stations

Lemma IV.3 proves that no behaviors in $B^1_{1,1}(i,j)$ may contain two informative events of s_i, s_j, which are synchronous, i.e., there cannot be any timestamp for which both s_i and s_j present an informative event. This implies that the maximum achievable throughput across such a buffer is 0.5, which may be considered suboptimal. Instead, buffer $B^2_{1,1}(i,j)$ is the minimum capacity buffer that is able to "transfer" one informative unit per timestamp, thus allowing, in the best case, to communicate with maximum throughput equal to one. Fig. 6 compares two possible behaviors of these buffers.

Lemma IV.3: $B^2_{1,1}(i,j)$ is the minimum capacity buffer with $l_f = l_b = 1$ s.t. for all K, closed intervals of $\mathbb{N}$

$$\begin{aligned}&\exists b^K = \left(s^K_1, \ldots, s^K_N\right) \in B^2_{1,1}(i,j) \wedge \forall k \in K,\\ &\left(e_k\left(s^K_i\right) \in \mathcal{E}_\iota\left(s^K_i\right) \wedge e_k\left(s^K_j\right) \in \mathcal{E}_\iota\left(s^K_j\right)\right). \quad (10)\end{aligned}$$

Proof: Relation (10) says that $B^2_{1,1}(i,j)$ is the minimum capacity buffer with $l_f = l_b = 1$ containing a behavior b^K, where s_i and s_j present $|K|$ consecutive pairs of synchronous informative events (i.e., the two informative events of each a pair have the same timestamp t_k) for all K, closed intervals of $\mathbb{N}$. Notice that the only buffer with $l_f = l_b = 1$ having capacity less than $B^2_{1,1}(i,j)$ is $B^1_{1,1}(i,j)$. We first show that $B^2_{1,1}(i,j)$ contains at least one behavior b^K satisfying relation (10) and then we prove that the same is not true for any behavior of $B^1_{1,1}(i,j)$. It is easy to construct an example of such a behavior for any K. For instance, consider a behavior $b = (s_1, \ldots, s_N)$ s.t. $\sigma(s^K_i) = \iota_1\ \iota_2 \ldots \iota_K\ \tau\ \tau\ \tau \ldots$ and that $\sigma(s^K_j) = \tau\ \iota_1\ \iota_2 \ldots \iota_K\ \tau\ \tau\ \tau \ldots$ Clearly, $s^K_i \equiv_\tau s^K_j$ and inequalities (4) and (5) (with $c = 2$) are satisfied for any $k \in K$. Hence, $b \in B^2_{1,1}(i,j)$. Moreover, at all timestamps $t_1, t_2, \ldots, t_K$, both s^K_i and s^K_j present an informative event.

Now, consider $B^1_{1,1}(i,j)$. If $c = 1$, combining inequalities (4) and (5), we obtain that $\forall k \in \mathbb{N}$

$$\begin{aligned}&\left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-1)}]}(s_j)\right]\right| + 1\\ &\quad \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_k]}(s_i)\right]\right| \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k+1)}]}(s_j)\right]\right|\\ &\left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k-1)}]}(s_i)\right]\right|\\ &\quad \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_k]}(s_j)\right]\right| \geq \left|\mathcal{F}_\iota\left[\sigma_{[t_0, t_{(k+1)}]}(s_i)\right]\right| - 1.\end{aligned}$$

$$\begin{array}{lcl} \sigma(s_1) & = & \iota_2\,\tau\;\iota_1\,\iota_3\,\iota_1\,\tau\;\iota_3\,\tau\;\tau\;\ldots \\ \sigma(s_2) & = & \tau\;\iota_8\,\tau\;\iota_4\,\tau\;\iota_7\,\iota_8\,\tau\;\iota_8\;\ldots \\ \sigma(s_3) & = & \tau\;\iota_6\,\tau\;\iota_5\,\iota_5\,\tau\;\iota_9\,\tau\;\iota_6\;\ldots \end{array} \quad \rightarrow \quad \begin{array}{lcl} \sigma(s_4) & = & \tau\;\iota_2\,\tau\;\iota_1\,\tau\;\iota_3\,\iota_1\,\tau\;\iota_3\;\ldots \\ \sigma(s_5) & = & \tau\;\iota_8\,\tau\;\iota_4\,\tau\;\iota_7\,\iota_8\,\tau\;\iota_8\;\ldots \\ \sigma(s_6) & = & \tau\;\iota_6\,\tau\;\iota_5\,\tau\;\iota_5\,\iota_9\,\tau\;\iota_6\;\ldots \end{array}$$

Fig. 7. Example of a behavior of an equalizer E with $I = \{1, 2, 3\}$ and $O = \{4, 5, 6\}$.

Hence, for all behaviors $b = (s_1, \ldots, s_N) \in B^1_{1,1}(i,j)$, signals s_i, s_j are not only latency equivalent, but also correlated according to a very regular pattern (see Fig. 6), which can be summarized in two properties: 1) there are no two synchronous informative events in s_i, s_j and 2) for all timestamps, informative events appear alternately on s_i and on s_j (possibly, at not consecutive timestamps). Property 1 is a negation of relation (10). □

Definition IV.3: The buffer $B^2_{1,1}$ is called a *relay station.*

V. Latency-Insensitive Design

In this section, we formally present the notion of latency-insensitive design as an application of the concepts previously introduced. To do so, we assume that:

1) the predesigned functional modules are synchronous functional processes;
2) the processes are strictly causal;
3) the processes belong to a particular class of processes called *stallable,* a weak condition to ask the processes to obey.

The basic ideas are as follows. Composing a set of predesigned synchronous functional modules in the most efficient way is fairly straightforward if we assume that the synchronous hypothesis holds. This composition corresponds to a composition of strict processes since there is *a priori* no need of inserting stalling events. However, as we have argued in the introduction, it is very likely that the synchronous hypothesis will not be valid for communication. If indeed the processes to be composed are patient, then adding an appropriate number of relay stations yields a process that is latency equivalent to the strict composition. Hence, if we use as the definition of correct behavior the fact that the sequences of informative events do not change, the addition of the relay stations solves the problem. However, requiring processes to be patient at the onset is definitely too demanding from a practical point of view. Still, in practice, a patient system can be *derived* from a strict one as follows: first, we take each strict process P_m and we compose it with a set of auxiliary processes to obtain an equivalent patient process P'_m. To be able to do so, all processes P_m must satisfy a simple condition (the processes must be stallable) that is formally specified in the next section. Then, we put together all patient processes by connecting them with relay stations. The set of auxiliary processes implements a "queuing mechanism" across the signal of P_m in such a way that informative events are buffered and reordered before being passed to P_m: informative events having the same ordinal are passed to P_m synchronously.

In the sequel, we first introduce the formal definition of functional processes. Then, we present the simple notion of stallable processes and we prove that every stallable process can be encapsulated into a wrapper process which acts as an interface toward a latency-insensitive protocol.

A. Stallable Processes

In the sequel, we consider only strictly causal processes and for each of them we assume that the well-founded order $\leq_c$ of Definition III.12 subsumes the causality relations among its signals, i.e., formally $\forall i \in I, \forall j \in O, (s_i \leq_c s_j)$.

Definition V.1: A process P with $I = \{1, \ldots, Q\}$ and $O = \{Q+1, \ldots, N\}$ is stallable iff for all its behaviors $b = (s_1, \ldots, s_Q, s_{Q+1}, \ldots, s_N) \in P$ and for all $k \in \mathbb{N}$

$$\forall i \in I \quad \left(\sigma_{[t_k,t_k]}(s_i) = \tau\right) \Leftrightarrow \forall j \in O \left(\sigma_{[t_{k+1},t_{k+1}]}(s_j) = \tau\right)$$

Hence, while a patient process tolerates arbitrary distributions of stalling events among its signals (as long as causality is preserved), a stallable process demands more regular patterns: τ symbols can only be inserted synchronously (i.e., with the same timestamp) on all input signals and this insertion implies the synchronous insertion of τ symbols on all output signals at the following timestamp. To assume that a functional process is stallable is quite reasonable with respect to a practical implementation. In fact, most hardware systems can be stalled: for instance, consider any sequential logic block that has a *gated clock* or a Moore finite state machine M with an extra input, which, if equal to τ, forces M to stay in the current state and to emit τ at the next cycle.

B. Encapsulation of Stallable Processes

Our goal is to define a group of functional processes that can be composed with a stallable process P to derive a patient process which is latency equivalent to P. We start considering a process that aligns all the informative events across a set of channels.

Definition V.2: An *equalizer* E is a process with $I = \{1, \ldots, Q\}$ and $O = \{Q+1, \ldots, 2 \cdot Q\}$, such that for all behaviors $b = (s_1, \ldots, s_Q, s_{Q+1}, \ldots, s_{2\cdot Q}) \in E$, we have that $\forall i \in I, (s_i \equiv_\tau s_{Q+i})$ and $\forall k \in \mathbb{N}$:

1) $\forall i, j \in O((\sigma_{[t_k,t_k]}(s_i) = \tau) \Rightarrow (\sigma_{[t_k,t_k]}(s_j) = \tau))$;
2) $\min_{i \in I}\{|\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]|\} - \max_{j \in O}\{|\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]|\} = 0$.

The first relation forces the output signals to have stalling events only synchronously, while the second guarantees that at every timestamp the number of informative events occurred at any output is always equal to the least number of informative events seen by any input signal up to that timestamp. Fig. 7 illustrates a possible behavior of an equalizer. Notice how the the presence of a stalling event at a certain input at a given timestamp does not necessarily force the presence of a stalling event on all outputs at the same timestamp. For instance, while the two stalling events on s_2 and s_3 at timestamps t_1 do force stalling events on all output signals at t_1, instead the stalling event present on s_1 at t_2 does not result in any timestamp on the output signals: this is due to the fact that at timestamp t_2, all input signals have seen

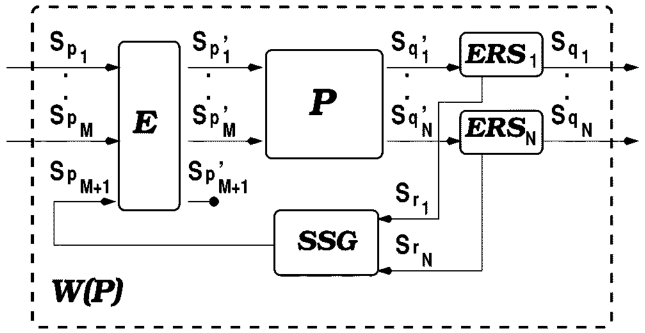


Fig. 8. Encapsulation of a stallable process P into a wrapper $W(P)$.

at least one informative event while no output event have been occurred on the output signal up to t_1.

Definition V.3: An *extended relay station* $\mathcal{ERS}$ is a process with $I = \{i\}$ and $O = \{j, l\}, i \neq j \neq l$ s.t. signals s_i, s_j are related by inequalities (2) and (3) of Definition IV.2 (with $l_f = l_b = 1$ and $c = 2$) and $\forall k \in \mathbb{N}$

$$\sigma_{[t_k,t_k]}(s_l) = \begin{cases} 1, & \text{if } \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_k]}(s_i)\right]\right| - \left|\mathcal{F}_\iota\left[\sigma_{[t_0,t_{k-1}]}(s_j)\right]\right| = 2 \\ 0, & \text{otherwise} \end{cases}.$$

Definition V.4: A *stalling signal generator* $\mathcal{SSG}$ is a process with $I = \{1, \ldots, Q\}$ and $O = \{Q+1\}$ s.t. $\forall b = (s_1, \ldots, s_{Q+1}), \forall k \in \mathbb{N}, \forall i \in [1, Q], (\mathcal{F}_\iota[\sigma_{[t_k,t_k]}(s_i)] \in [0,1])$ and

$$\sigma_{[t_k,t_k]}(s_{Q+1}) = \begin{cases} \tau, & \text{if } \exists j \in [1,Q]\left(\mathcal{F}_\iota\left[\sigma_{[t_k,t_k]}(s_j)\right] = 1\right) \\ 0, & \text{otherwise} \end{cases}.$$

As illustrated in Fig. 8, any stallable process P can be composed with an equalizer, a stalling signal generator, and some extended relay stations to derive a patient process that is latency equivalent to P.

Definition V.5: Let P be a stallable process with $I_P = \{p'_1, \ldots, p'_M\}$ and $O_P = \{q'_1, \ldots, q'_N\}$. A *wrapper process* (or *shell process*) $W(P)$ of P is the process with $I_W = \{p_1, \ldots, p_M\}$ and $O_W = \{q_1, \ldots, q_N\}$, which is obtained by composing P with the following processes:

1) an equalizer E with $I_E = \{p_1, \ldots, p_M, p_{M+1}\}$ and $O_E = \{p'_1, \ldots, p'_M, p'_{M+1}\}$;
2) N extended relay stations $\mathcal{ERS}_1, \mathcal{ERS}_2, \ldots, \mathcal{ERS}_N$ s.t. $I_j = \{q'_j\}$ and $O_j = \{q_j, r_j\}$, with $j \in [1, N]$
3) a *stalling signal generator* $\mathcal{SSG}$ with $I_G = \{r_1, \ldots, r_N\}$ and $O_G = \{p_{M+1}\}$.

Theorem V.1: Let $W(P)$ be the wrapper process of Definition V.5. Process $W = \text{proj}_{I_W \cup O_W}(W(P))$ is a patient process that is latency equivalent to P.

Proof: Throughout the proof, we follow the index notation of Definition V.5.

$W \equiv_\tau P$: Let $b' = (s_{p'_1}, \ldots, s_{p'_M}, s_{q'_1}, \ldots, s_{q'_N})$ be a behavior of P and $b = (s_{p_1}, \ldots, s_{p_{M+1}}, s_{q_1}, \ldots, s_{q_N}, s_{p'_1}, \ldots, s_{p'_{M+1}}, s_{q'_1}, \ldots, s_{q'_N}, s_{r_1}, \ldots, s_{r_N})$ one of $W(P)$. Let $b_W = \text{proj}_{I_W \cup O_W}(b) = (s_{p_1}, \ldots, s_{p_M}, s_{q_1}, \ldots, s_{q_N})$ be the corresponding behavior of W. Then, by Definition V.2 of equalizer $\forall i \in I_E, (s_i \equiv_\tau s'_i)$, and, by definition of relay station, $\forall i \in [1, N], (s_{q_i} \equiv_\tau s_{q'_i})$. Therefore, $b_W \equiv_\tau b$.

W Patient: Recalling Definition III.16, we need to prove that $\forall b = (s_{p_1}, \ldots, s_{p_M}, s_{q_1}, \ldots, s_{q_N}) \in W, \forall j \in I_W \cup O_W, \forall e_k(s_j) \in \mathcal{E}_\iota(s_j)$

$$(\mathcal{PE}[\text{stall}(e_k(s_j))] \cap W \neq \emptyset)$$

Consider first stalling any input signal of W: for all $s_{p_i}, i \in [1, M]$ and all $e_g(s_{p_i}) \in \mathcal{E}_\iota(s_{p_i})$, let $b' = (s'_{p_1}, \ldots, s'_{p_M}, s'_{q_1}, \ldots, s'_{q_N}) = \text{stall}(e_g(s_{p_i}))$. Two cases may happen.

1) There exists a signal $s_{p_k}, k \in [1, M], k \neq i$, s.t. $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_k})]| < |\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_i})]|$. As a consequence, by Definition V.2, no additional stalling events are added at the output of E nor, ultimately, on the output signals of W. Hence, while stall move $\text{stall}(e_g(s_{p_i}))$ does not affect any other signal, but s_{p_i}, still $b' \in W$ and $\mathcal{PE}[\text{stall}(e_k(s_{p_i}))] = b'$ (the stall move is "absorbed" by the equalizer). Therefore, $\mathcal{PE}[\text{stall}(e_k(s_{p_i}))] \cap W \neq \emptyset$.
2) s_{p_i} is a signal of b s.t. $|\mathcal{F}_\iota[\sigma_{[t_0,t_g]}(s_{p_i})]| = \min_{i \in I}\{|\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]|\}$. In this case, the insertion of a stalling event on s_{p_i} at t_g implies that all the output signals of equalizer E have a stalling event at t_g. Then, by analyzing the interrelationships among the components of W, it is easy to verify that $b' \notin W$. In fact, all the output signals of P are forced to have a stalling event at t_{g+1} and, similarly, all the output signals $s_{q_1}, \ldots, s_{q_N}$ to have it at t_{g+2}. Hence, $\forall j \in N, e_{g+2}(s_{q_j}) \in \mathcal{E}_\iota(s_{q_j})$ must be also stalled. Then, since move $\text{stall}(e_g(s_{p_i}))$ does not affect any other signal, but $s_{p_i}, b' \notin W$. However, since $\text{ord}(e_{g+2}(s_{q_j})) = \text{nextEvent}((s_{q_j}), e_g(s_{p_i}))$, the insertion of one stalling event on each of the wrapper outputs at t_{k+2} is compatible with the definition of procrastination effect and, therefore, $\mathcal{PE}[\text{stall}(e_k(s_{p_i}))] \cap W \neq \emptyset$.

Next, consider stalling any output signal of W: for all $s_{q_j}, j \in [1, N]$ and all $e_h(s_{q_j}) \in \mathcal{E}_\iota(s_{q_j})$, let $b' = (s'_{p_1}, \ldots, s'_{p_M}, s'_{q_1}, \ldots, s'_{q_N}) = \text{stall}(e_h(s_{q_j}))$. By definition of stall move, $\forall m \in [1, M], (s'_{p_m} = s_{p_m})$ and $\forall n \in [1, N], n \neq j, (s'_{q_n} = s_{q_n})$. Hence, again, $b' \notin W$. In fact, the insertion of a stalling event on signal s_{q_j} at t_h has an impact on signal $s_{q'_j}$ of $\mathcal{ERS}_j$, which is *constrained* to stall the input event $e_{h+l}(s_{q'_j}) \in \mathcal{E}_\iota(s_{q'_j})$ occurring l timestamps later.[16] As a consequence, all the outputs of the stallable process P must have a stalling event at t_{h+l}. While no other stalling events are forced on s_{q_j} of $\mathcal{ERS}_j$ at t_{h+l+1}, all the remaining relay stations $\mathcal{ERS}_r, r \in [1, N]/\{j\}$ must stall their $e_{h+l+1}(s_{q_r}) \in \mathcal{E}_\iota(s_{q_r})$. Hence, $b' \notin W$ because $\text{stall}(e_h(s_{q_j}))$ does not affect any other signal, but s_{q_j}. However, $\forall r \in ([1, N]/\{j\})$, since $e_{h+l+1}(s_{q_r}) = \text{nextEvent}(s_{q_r}, e_g(s_{q'_r}))$, where $e_g(s_{q'_r}) = \text{nextEvent}(s_{q'_r}, e_h(s_{q_j}))$, then $e_h(s_{q_j}) \leq_{\text{lo}} e_{h+l+1}(s_{q_r})$. Hence, the insertion of one stalling event on each wrapper output s_{q_r}, $r \in ([1, N]/\{j\})$ at t_{h+l+1} is compatible with the definition of procrastination effect. Therefore, $\mathcal{PE}[\text{stall}(e_h(s_{q_i}))] \cap W \neq \emptyset$. □

[16]Note that, by Definition IV.3, $e_{h+l}(s_{q'_j})$ must be stalled even though $\forall k \in [h+1, h+l-1], (\sigma_{[t_k,t_k]}(s_{q'_j}) = \tau)$

C. Latency-Insensitive Design Methodology

By putting together the ideas discussed in the previous sections, we can derive the following guidelines for the definition of the new design methodology.

1) Begin with a system of stallable processes and channels.
2) Encapsulate each stallable process to yield a corresponding wrapper process that is patient and latency equivalent to the original one.
3) By inserting the required amount of relay stations on each channel, the latency of the communication among any pair of processes now can be arbitrarily varied without affecting the overall system functionality.

This approach clearly "orthogonalizes" computation and communication: in fact, we can build systems by assembling functional cores (which can be arbitrarily complex as far as they satisfy the stalling assumption) and wrappers (which interface them with the channels, by "speaking" the latency-insensitive protocol). While the specific functionality of the system is distributed in the cores, the wrappers can be automatically generated around them.[17] Furthermore, the validation of the system now can be efficiently decomposed based on assume-guarantee reasoning and compositional model checking [35]–[38]: each wrapper is verified assuming a given protocol and the protocol is verified separately.

With regard to the design of digital integrated circuits, the theory of latency-insensitive design can be used as the formal basis for defining the *latency-insensitive design methodology* that is centered around the following simple noniterative design flow.

1) The designers design and validate the chip as a collection of synchronous modules that can be specified with the usual hardware-description languages (HDLs).
2) Each module is automatically encapsulated within a block of control logic (the wrapper) to make it latency insensitive.
3) Traditional logic synthesis and place-and–route steps are applied.
4) If the presence of unexpectedly large wire delays makes it necessary, the resulting layout is corrected by simply inserting the right amount of relay stations to meet the clock cycle constraints everywhere.

Notice that the design, layout, and routing of individual modules would not need to be changed to reflect any necessary changes in wire latencies during the chip-level layout and wiring process. This may clearly represent a significant advantage for future system-on-a-chip designs, where the designers completing the chip-level integration most likely will not work at the same company as the designers of the individual modules. Furthermore, since it is based on the synchronous assumption, the approach facilitates the adoption of state-of-the art formal verification techniques within a new design flow that, for the rest, can be built using traditional and well-known layout and synthesis CAD tools.

[17]This is the reason why wrappers are also called *shells*: they just "protect" the IP (*the pearl*) they contain from the "troubles" of the external communication architecture.

D. Impact on System Performance

Naturally, the effectiveness of the latency-insensitive design methodology is strongly related to the ability of maintaining a sufficient communication throughput in the presence of increased channel latencies. However, in the case of integrated circuit design, this is just one instance of a more general problem that has to be faced while using DSM technologies. In fact, since on-chip communication was not an issue with previous process technologies, the vast majority of chips that have been realized over the past two decades are based on architectural models relying on low-latency communication to shared global resources. The advantages of such models is that they provide the most uniform computational framework and the best utilization of the functional units. On the other hand, this focus on function rather than communication is now seen as the fundamental conceptual roadblock to be overcome in DSM design [7]. In this regard, the latency-insensitive design methodology represents an interesting approach due to the inherent separation of communication and computation.

Inserting extra latency stages on a cyclic pipeline does not necessary translate into a performance hit. For instance, the case of the Alpha 21264 microprocessor, where the integer unit is partitioned into two modules and the latency for communicating between them takes an additional clock cycle [39], shows how it is possible to pipeline long wires, thereby increasing their latency, while still offering high computational bandwidth. Similarly, the presence of so-called *drive stages* in the new hyperpipelined NETBURST microarchitecture [12] suggests that even for high-end designs, such as the Pentium 4 microprocessor, the insertion of extra stages dedicated exclusively to handle wire delays may be the result of a precise engineering choice. From this point of view, the present paper represents a formal background for the definition of design methodologies that allow an efficient analysis and exploitation of the latency/throughput tradeoffs at any level of the design flow.

Furthermore, the latency-insensitive approach can be extended to incorporate other techniques aimed to have the performance of a design less susceptible to large variations in channel latency. For instance, a simple technique to make the design more robust on this respect consists in ensuring that the design specification contains some "slack" in the form of unneeded pipeline delays. If there is sufficient "slack" latency around every cyclic path in the design, then, after the final layout is derived, the latency-insensitive protocol allows us to distribute this slack in a completely transparent manner to cover an increase in wire latency without any changes to the design or the layout of the modules (a sort of *dynamic retiming*). The overall design becomes more robust in the sense that it is less likely that some modules would have to be changed to recover performance lost due to unexpectedly large wire delays. Nonetheless, there will be cases where cyclic paths of low latency affecting the overall system throughput are inevitable. In such cases, techniques typically used for microprocessor design such as *speculation* and *out-of-order execution* may be embodied within a latency-insensitive protocol. For instance, when a particular data item in not yet computed, while being on the critical path, it is sometimes possible to "guess" the likely value of this data and to allow the computation to proceed. Later, this value can be "retracted" if it proves to be incorrect. Such techniques may dramatically increase the overall system performance, but they are currently adopted only in high-end microprocessor design because they are error-prone and quite difficult to implement. However, their demand is destined to grow, as wire latencies will keep increasing. On the other hand, these techniques may be rigorously built into a latency-insensitive protocol that would allow speculation to break the tight dependency cycles (provided the designer can identify an appropriate guessing strategy). Being part of the protocol, they would be isolated from the functional specification and they would enter the design picture in a *correct-by-construction* manner that would not risk the introduction of a design error or cause previous simulation work to be invalidated.

VI. Case Study—The PDLX Microprocessor

To experiment the proposed methodology, we performed a "latency-insensitive design" of PDLX, an out-of-order microprocessor with speculative execution. In the present section, we first summarize the architectural specification of PDLX, and, then, we discuss the latency-insensitive design as well as the experimental results.

A. PDLX Architecture and Instruction Flow

The instruction set of PDLX is the same as the one of the DLX microprocessor, described in [11]. The PDLX architecture is based on an extended version of the *Tomasulo's algorithm* [40], which combines traditional dynamic scheduling with hardware-based speculative execution. As a consequence, the data path of PDLX is similar to the one of some of the most advanced microprocessor available on the market today [39], [41].

The PDLX architecture is conceptually illustrated by the block diagram of Fig. 9. At the center of the diagram lies a set of execution units (gray shaded) that operate in parallel. Schematically, the PDLX behavior can be summarized as follows: the *branch processing unit* sends the next value of the *program counter (PC)* to the *fetch unit*, which fetches the corresponding instruction from the *instruction cache* and passes it to the *decode unit*. Once instruction decoding is completed, the result arrives to the *dispatch unit*, which interacts with the *reservation stations (RSs)* of the different processing units as well as with the *completion unit* and the *system register unit*. Instructions are fetched, decoded, and dispatched sequentially following the order of the program that gets executed. Once a new decoded instruction I arrives, the *dispatch unit* starts by assigning it to one of the functional categories (integer arithmetic/logic, floating-point, load, store, branch, ...) and then checks whether the following two conditions are verified.

1) There is one entry available in the reorder buffer within the *completion unit*.
2) There is an available reservation station at the head of a processing unit matching the function category to which the instruction belongs.

if one of these conditions is not satisfied the *dispatch unit* stalls, otherwise instruction I is dispatched: this means that the instruction is labeled with a *tag* t_I identifying the reorder buffer entry which has been assigned to it and, at the end of the execution, will contain the result. Then, operands and opcode of instruction I are loaded into the selected reservation station to start the execution. In fact, the execution starts immediately only if the values of the operands which have been read from the *system register unit* are "currently correct" in the sense that no other instruction I' (previously dispatched and still not completed) is destined to change the value of one of these operands. If this is indeed the case, for each operand whose value is not yet available the *dispatch unit* writes on the corresponding entry in the reservation station the tag $t_{I'}$ that identifies the reorder buffer entry previously assigned to instruction I' on which the operand depends. The execution of instruction I is procrastined until all correct values of its operands arrive at the reservation station. Different instructions may take a different number of clock cycles to execute not only due to this procrastination, but also because their executions may present different latencies. When the execution of an instruction ends, the corresponding processing unit broadcast the result to the reorder buffer entry and to any reservation station that is awaiting it.

The *completion unit* tracks each instruction from dispatch through execution and retires it by removing the result from the bottom entry of the reorder buffer and committing it to update the system registers. In-order completion guarantees that the system is in the correct state when it is necessary to recover from a mispredicted branch or any exception. In the presence of a conditional branch whose condition cannot be resolved immediately, the *branch processing unit* predicts the "branch target address" and instruction fetching, dispatching and execution continue from the predicted path. However, these instructions can not commit and write back results into the system register until the prediction has been resolved, i.e., determined to be correct. Instead, when a prediction is determined to be incorrect, the instruction from the wrong path are flushed from the datapath and the execution resumes from the correct path.

B. Latency-Insensitive Design of PDLX

We performed a high-level cycle-accurate design of PDLX by using BONeS DESIGNER [42], a CAD tool that provides a powerful modeling and analysis environment for system design. We first defined a synchronous specification of the PDLX and we designed each of of the PDLX modules illustrated in Fig. 9, keeping in mind only the following informal rule to make the process stallable: *at each clock cycle, the execution process of a module can always be frozen without affecting its internal state*. Independently from the design of the PDLX modules, we specified also a latency-insensitive protocol library, which contains parameterized components to guide the automatic generation of different kinds of relay stations and wrappers. Finally, we encapsulated each module in a wrapper and we obtained the final system. Obviously, this decomposition of the hardware implementing the PDLX is not the only possible, let alone the best, one. Still, while reasonably simple, it presents interesting challenges to the realization of the proposed latency-insensitive communication architecture. In particular, modules such as the *fetch unit* and the *dispatch unit* merge channels coming from separate sources, and which are likely to have different latencies. Further, each time the predicted value of a conditional

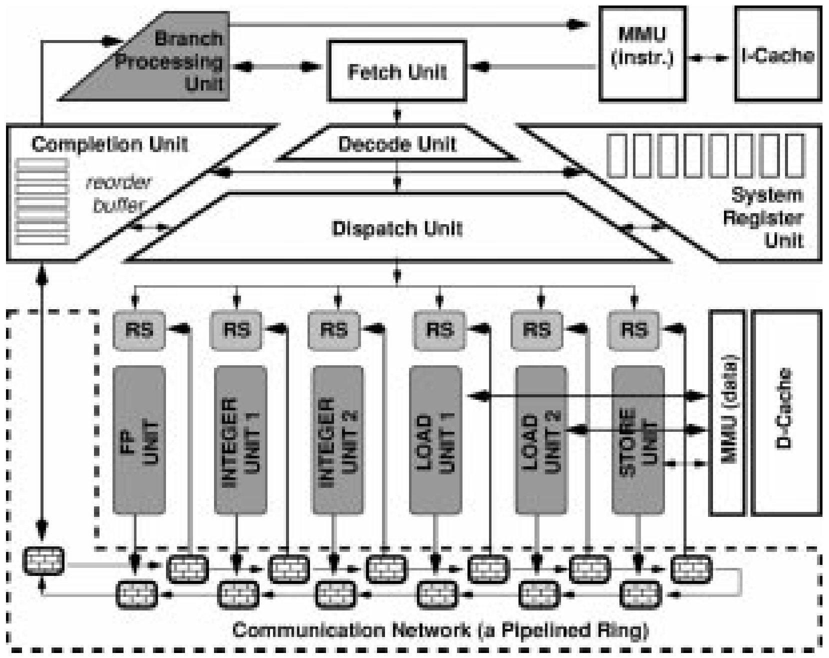


Fig. 9. PDLX microprocessor block diagram: conceptual view.

branch is verified a "feedback path" gets activated from the system register through the *completion unit* to the *branch processing unit*.

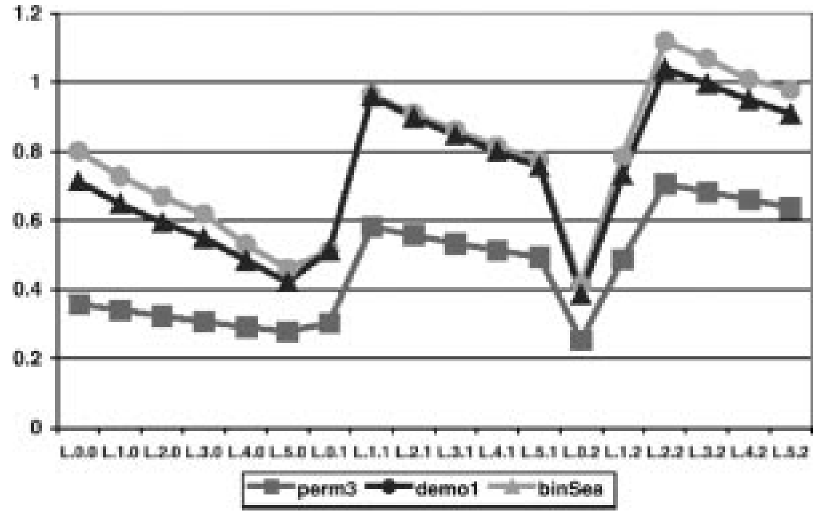
We specified most PDLX modules based on the assumption that they communicate by means of point-to-point channels, whose latencies may be arbitrary. However, due to the particular structure of a microprocessor such as PDLX and, in particular, to the parallel organization of its execution units, we decided to adopt a different type of communication structure to connect the several relay stations, the execution units and the reorder buffer: a *pipelined ring*. A ring, like a bus, inherently supports broadcast-based communication: the sender place a packet on the ring while the other modules inspect (*snoop*) it as it goes by to see if it is relevant to them [43]. In our case, the snooping mechanism is obviously based on identifying the tags associated to the entries of the *Reorder Buffer*, as described in Section VI-A. In general, to keep sequential consistency on a ring is more complex than on a bus since multiple packets may traverse it simultaneously. However, the characteristics of our system help us in this perspective because the *completion unit* guarantees in-order instruction commitment together with the correct serial progress of the state of the system. The linear point-to-point nature of a ring allows aggressive pipelining and potentially very high clock rates. A disadvantage is that the communication latency is high, growing linearly with the number of nodes on the ring. Overall, based on its characteristics, a ring represents an interesting design solution in the framework of our design methodology, where modules are by definition latency insensitive and pipelining is extensively applied.

C. Experimental Results

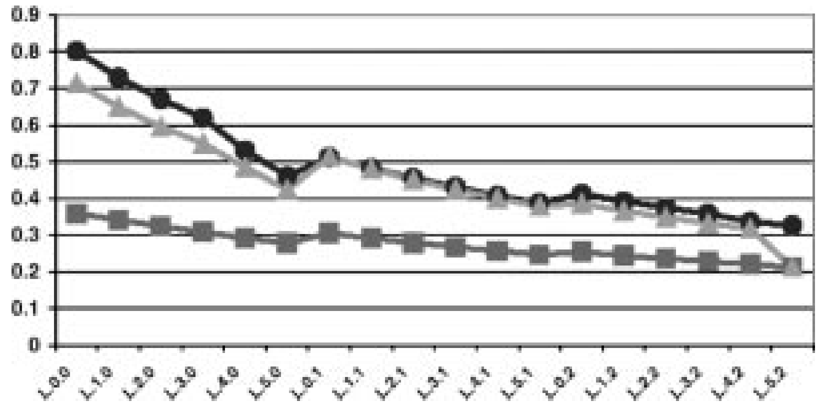
To test our design, we took some simple numerical C programs (permutations, binary search, ...) and we generated the corresponding DLX assembler code by using DLXCC, a publicly available DLX compiler [44]. Then, we loaded the assembler into the PDLX *instruction cache* and we executed it while logging every read/write access to the *data cache*. Finally, we compared the "log file" with the one obtained executing the same assembler code on the DLX simulator DLXSIM to verify that the functional behavior was indeed the same.

For each program execution, we computed the total number of clock cycles N necessary to complete the execution of the assembler code: this number is equal to $I+S+P$, where I is the number of instruction that have been issued, S is the number of cycles lost due to a stall within the execution unit, and P is the number of cycles lost due to pipeline latency. Since the PDLX is a single-issue multiprocessor, the instruction throughput $T = I/N$ is a quantity less than or equal to one. This quantity can be multiplied by the system clock frequency to obtain the *effective instruction throughput* $ET = (I/N) * f_{\text{CLK}}$, which allows us to compare the execution of the same assembler code on different PDLX implementations running at different speeds. Fig. 10 illustrates the results obtained running three different assembler programs: on the y axis, we have the instruction throughput on the bottom chart and the effective instruction on the top chart. In both charts, each discrete point on the x axis corresponds to a different PDLX implementation with a different fixed amount of latency on some communication channels.

For this experiment, we focused on two specific channels on Fig. 9: channel C_a between the *instruction memory management unit* and the *instruction cache (I-Cache)* and channel C_b between the *data memory management unit* and the *data cache (D-Cache)*. We varied the latencies of the two channels as follows: going from left to right on the x axis, each of the 18 data points represents an implementation case and is labeled as L_a_b, where a and b denote the amounts by which the latencies of channels C_a and C_b have been increased. In particular, we varied a from zero to five and b from zero to two. As expected, the bottom chart confirms that the more we increase the latencies



(a)



(b)

Fig. 10. PDLX performance. (a) Throughput. (b) Effective throughput.

between the two caches and the rest of the system, the higher is the throughput degradation. It is also clear that for this PDLX implementation, the impact of increasing the D-Cache latency by one unit while leaving untouched the I-Cache latency (data point L_0_1) is more or less equivalent to increase the I-Cache latency by four units while leaving untouched the D-Cache latency (data-point L_4_0)

The data illustrated in the above chart of Fig. 10 have been obtained based on the assumption that the wires grouped in channels C_a and C_b represent the critical path of the overall PDLX design and that, after segmenting them (by inserting relay stations), we could afford to raise the clock frequency appropriately. Specifically, for each implementation case, we set the system clock cycle as $f_{\mathrm{CLK}} = \min\{a, b\} + 1$. One could argue that the assumption is too coarse because, for instance, it is unlikely that all the other modules in the design are able to work correctly after doubling the clock. However, the main point that we want to stress here is that within the present methodology, one may perform an early exploration of the latency/throughput tradeoffs to guide architectural choices based on a rough estimation of the channel latencies and then keep refining these choices at the different stages of the design flow to accommodate various implementation constraints, while relying always on the property of the latency-insensitive communication protocol. In this regard, it is important to emphasize that all the above implementations are functionally equivalent by construction, being obtained simply by changing the number of relay stations on the channels and with no need of redesigning any PDLX module. Furthermore, the insertion of relay stations can be made at late stages in the design process, after detailed information can be extracted from the physical layout, to "fix" those channels whose latencies are longer than the desired clock cycle. While performing this operation, it is easy to keep an exact track of the throughput variations.

VII. Conclusion

This paper presents the theory of latency-insensitive design. Latency-insensitive designs are synchronous distributed systems composed by functional modules that exchange data on communication channels according to a latency-insensitive protocol. The protocol guarantees that latency-insensitive systems, composed of functionally correct modules, behave correctly independently of the channel latencies. This allows us to increase the robustness of a design implementation because any delay variations of a channel can be "recovered" by changing the channel latency while the overall system functionality remains unaffected. The protocol works on the assumption that the functional modules are stallable, a weak condition to ask the processes to obey.

An important application of the proposed theory is represented by the latency-insensitive methodology to design large digital integrated circuits with DSM technologies. The methodology is based on the assumption that the design is built by assembling blocks of IPs that have been previously designed and verified. Thanks to the compositionality of the notion of latency equivalence, this methodology allows us to orthogonalize communication and computation, while the timing requirements imposed by the clock are met by construction. Furthermore, since it is based on the synchronous assumption, the approach facilitates the adoption of formal validation techniques within a new design flow that, for the rest, can be built using traditional CAD tools.

Acknowledgment

The authors would like to thank L. Lavagno and A. Saldanha for the discussions that led to the theory of latency-insensitive design. They would also like to thank P. Scaglia for his support and continuous encouragement. Finally, they would also like to thank the reviewers for their help in improving the overall quality of the paper.

References

[1] J. Cong, "Challenges and opportunities for design innovations in nanometer technologies," SRC Design Sciences Concept Paper, Dec. 1997.

[2] H. Kapadia and M. Horowitz, "Using partitioning to help convergence in the standard-cell design automation method," in *Proc. Design Automation Conf.*, June 1999, pp. 592–597.

[3] M. T. Bohr, "Interconnect scaling—The real limiter to high performance ULSI," in *Proc. IEEE Int. Electron Devices Meeting*, Dec. 1995, pp. 241–244.

[4] R. Ho, K. Mai, H. Kapadia, and M. Horowitz, "Interconnect scaling implications for CAD," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1999, pp. 425–429.

[5] M. T. Bohr, "Silicon trends and limits for advanced microprocessors," *Commun. ACM*, vol. 41, no. 3, pp. 80–87, Mar. 1998.

[6] M. J. Flynn, P. Hung, and K. W. Rudd, "Deep-submicron microprocessor design issues," *IEEE Micro*, vol. 19, pp. 11–13, July 1999.

[7] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[8] D. Matzke, "Will physical scalability sabotage performance gains?," *IEEE Comput.*, vol. 8, pp. 37–39, Sept. 1997.

[9] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conentional microarchitectures," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, June 2000, pp. 248–250.

[10] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981, Advanced Computer Science Series.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1996.

[12] P. Glaskowski, "Pentium 4 (partially) previewed," *Microprocessor Rep.*, vol. 14, no. 8, pp. 10–13, Aug. 2000.

[13] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *Proceedings of the 11th International Conference on Computer-Aided Verification*, N. Halbwachs and D. Peled, Eds. New York: Springer-Verlag, 1999, vol. 1633, pp. 123–133.

[14] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for 'correct-by-construction' latency insensitive design," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1999, pp. 309–315.

[15] A. Davis and S. M. Nowick, "Asynchronous circuit design: Motivation, background, and methods," in *Asynchronous Digital Circuit Design, Workshops in Computing*, G. Birtwistle and A. Davis, Eds. New York: Springer-Verlag, 1995, pp. 1–49.

[16] S. Hauck, "Asynchronous design methodologies: An overview," *Proc. IEEE*, vol. 83, no. 1, pp. 69–93, Jan. 1995.

[17] W. A. Clark, "Macromodular computer systems," in *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*. New York: Academic, 1967, vol. 30, pp. 335–336.

[18] W. A. Clark and C. E. Molnar, "The promise of macromodular systems," in *Proceedings of the 6th Annual IEEE Computer Society International Conference*. Piscataway, NJ: IEEE Press, 1972, pp. 309–312.

[19] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger, "Synthesis of delay-insensitive modules," in *1985 Chapel Hill Conference on Very Large Scale Integration*, H. Fuchs, Ed. Rockville, MD: Computer Science, 1985, pp. 67–86.

[20] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang, "Q-modules: Internally clocked delay-insensitive modules," *IEEE Trans. Comput.*, vol. 37, pp. 1005–1018, Sept. 1988.

[21] J. T. Udding, "A formal model for defining and classifying delay-insensitive circuits," *Distrib. Comput.*, vol. 1, no. 4, pp. 197–204, 1986.

[22] J. A. Brzozowski and J. C. Ebergen, "On the delay-sensitivity of gate networks," *IEEE Trans. Comput.*, vol. 41, pp. 1349–1360, Nov. 1992.

[23] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Advanced Research in VLSI: Proceedings of the 6th MIT Conference*, W. J. Dally, Ed. Cambridge, MA: MIT Press, 1990, pp. 263–278.

[24] J. C. Ebergen, "A formal approach to designing delay-insensitive circuits," *Distrib. Comput.*, vol. 5, no. 3, pp. 107–119, 1991.

[25] M. B. Josephs and J. T. Udding, "An overview of DI algebra," in *Proc. Hawaii International Conference on System Sciences*. Los Alamitos, CA: IEEE Comput. Soc. Press, 1993, vol. I.

[26] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," Ph.D. dissertation, California Inst. Technol., Pasadena, CA, 1991.

[27] P. Beerel and T. H.-Y. Meng, "Automatic gate-level synthesis of speed-independent circuits," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 581–587.

[28] D. L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," in *ACM Distinguished Dissertations*. Cambridge, MA: MIT Press, 1989.

[29] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev, "Basic gate implementation of speed-independent circuits," in *Proc. Design Automation Conf.*, June 1994, pp. 56–62.

[30] J. L. A. van de Snepscheut, *Trace Theory and VLSI Design*. Berlin, Germany: Springer-Verlag, 1985, vol. 200, Lecture Notes in Computer Science.

[31] D. C. Ku and G. De Micheli, "Relative scheduling under timing constraints," in *Proc. Design Automation Conf.*, June 1990, pp. 59–64.

[32] D. Filo, D. Ku, C. Coelho, and G. De Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 268–281, Sept. 1993.

[33] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1217–1229, Dec. 1998.

[34] A. Benveniste and P. L Guernic, "Hybrid dynamical systems theory and the signal language," *IEEE Trans. Automat. Contr.*, vol. 5, pp. 535–546, May 1990.

[35] E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional model checking," in *Proc. 4th Annu. Symp. Logic in Computer Science*, Asilomar, CA, June 1989, pp. 464–475.

[36] K. L. McMillan, "A compositional rule for hardware design refinement," in *Proc. 9th Int. Conf. Computer-Aided Verification*, Haifa, Israel, June 1997, pp. 24–35.

[37] ——, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *Proc. 10th Int. Conf. Computer-Aided Verification*, Vancouver, BC, Canada, July 1998, pp. 110–121.

[38] T. A. Henzinger, S. Qadeer, and R. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Proc. 10th Int. Conf. Computer-Aided Verification*, Vancouver, BC, Canada, July 1998, pp. 440–451.

[39] B. A. Gieseke *et al.*, "A 600 MHz superscalar RISC microprocessor with out-of-order execution," *Tech. Dig. IEEE Int. Solid-State Circuits Conf.*, pp. 176–177, Feb. 1997.

[40] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Devel.*, vol. 11, pp. 25–33, Jan. 1967.

[41] R. E. Kessler, "The alpha 21 264 microprocessor," *IEEE Micro.*, vol. 19, pp. 24–36, Mar. 1999.

[42] S. J. Schaffer and W. W. LaRue, "BONeS DESIGNER: A graphical environment for discrete-event modeling and simulation," in *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Los Alamitos, CA: IEEE Comput. Soc. Press, 1994, pp. 371–374.

[43] D. E. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Mateo, CA: Morgan Kaufmann, 1999.

[44] The DLX Software (1994). [Online]. Available: ftp://max.stanford.edu/pub/hennessy-patterson.software

Luca P. Carloni (S'95) received the Laurea degree (summa cum laude) in electrical engineering from the University of Bologna, Bologna, Italy, in 1995, and the M.S. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1997. He is currently working towards the Ph.D. degree in electrical engineering and computer sciences at the same university.

His current research interests include embedded systems design, high-level synthesis, logic synthesis, and combinatorial optimization.

Kenneth L. McMillan received the B.S. degree in electrical engineering from the University of Illinois, Urbana, in 1984, the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1986, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1992.

He has been a Chip Designer, Biomedical Engineer, a Member of the Technical Staff at AT&T Bell Laboratories, and is currently a Research Scientist at Cadence Berkeley Laboratories, Berkeley, CA. His current research interests include computer music, formal verification, and design methodology.

Alberto L. Sangiovanni-Vincentelli (F'83) received the Dott. Ing. degree (summa cum laude) in electrical engineering and computer science from the Politecnico di Milano, Milan, Italy, in 1971.

He holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences at the University of California, Berkeley, where he has been on the Faculty since 1976. From 1980 to 1981, he spent a year as a Visiting Scientist with Mathematical Sciences Department of the IBM T.J. Watson Research Center, Yorktown Heights, NY. In 1987, he was a Visiting Professor with the Massachusetts Institute of Technology, Cambridge. He cofounded Cadence Design Systems (where he is currently the Chief Technology Advisor and Member of the Board of Directors), Synopsys, Inc. (where he was Chair of the Technical Advisory Board), and Comsilica, a startup in the wireless communication area (where he is currently the Chairman of the Board). He also founded the Cadence Berkeley Laboratories and the Kawasaki Berkeley Concept Research Center, where he is Chairman of the Board. He was a Director of ViewLogic and Pie Design Systems. He is currently a Member of the Board of Directors of Sonics Inc., Softface, and Accent. He has consulted for a number of U.S. companies, including IBM, Intel, AT&T, GTE, GE, Harris, Nynex, Teknekron, DEC, HP, Japanese companies, including Kawasaki Steel, Fujitsu, Sony and Hitachi, and European companies, including SGS-Thomson Microelectronics, Alcatel, Daimler-Benz, Magneti-Marelli, BMW, Bull. He is the Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems, a European Group of Economic Interest. He is on the Advisory Board of the Lester Center of the Haas School of Business and of the Center for Western European Studies and a member of the Berkeley Roundtable of the International Economy. He has authored or coauthored over 530 papers and 14 books in the area of design methodologies, large-scale systems, embedded controllers, hybrid systems and tools.

Dr. Sangiovanni-Vincentelli is a Member of the National Academy of Engineering. He received the Distinguished Teaching Award of the University of California in 1981, the Guillemin-Cauer Award in 1982, the Darlington Award in 1987, and the 1995 Graduate Teaching Award of the IEEE. He was the Technical Program Chairperson of the International Conference on Computer-Aided Design and is currently General Chair and was also the Executive Vice-President of the IEEE Circuits and Systems Society.