INVITED PAPER

From Latency-Insensitive Design to Communication-Based System-Level Design

This paper overviews the principles and practice of latency-insensitive design, offers a retrospective on related research over the past decade, and looks ahead in proposing the protocols and shells paradigm as the foundation to bridge the gap between system-level and logic/physical design, a requisite to cope with the complexity of engineering future system-on-chip platforms.

By Luca P. Carloni, *Senior Member IEEE*

ABSTRACT | By the end of the 20th century, the continuous progress of the semiconductor industry brought a major transformation in the design of integrated circuits: as the speed of global wires could not keep up with the speed of ever-smaller transistors, the digital chip became a distributed system. This fact broke the synchronous paradigm assumption, i.e., the foundation of those computer-aided design (CAD) flows which had made possible three decades of unique technology progress: from chips with thousands of transistors to systems on chips (SoCs) with over a billion transistors. Latency-insensitive design (LID) is a correct-by-construction design methodology that was originally developed to address this challenge while preserving as much as possible the synchronous assumption. A broad new approach that transforms the fundamentals of how complex digital systems are assembled, LID introduces the protocols and shells paradigm, which offers several main benefits: modularity (by reconciling the synchronous paradigm with the dominant impact of global interconnect delays that characterizes nanometer technologies), scalability (by making key properties of the design be correct by construction through interface synthesis), flexibility (by simplifying the design and validation of a system through the separation of communication from computation), and efficiency (by enabling the reuse of predesigned components, thus reducing the overall design time). This paper overviews the principles and practice of LID, offers a retrospective on related research over the past decade, and looks ahead in proposing the protocols and shells paradigm as the foundation to bridge the gap between system-level and logic/physical design, a requisite to cope with the complexity of engineering future SoC platforms.

KEYWORDS | Computer engineering; computer-aided design (CAD); embedded systems; integrated circuits; latency-insensitive design (LID); system-level design (SLD); system on chip (SoC)

I. INTRODUCTION: A PARADIGM SHIFT?

Paradigms are "accepted examples of scientific practice—examples which include law, theory, application, and instrumentation together—[that] provide models from which spring particular coherent traditions of scientific research." This at least according to Kuhn in his classic 1962 book, a landmark event in the philosophy of science [1]. The informal definition is centered around the English word that best translates the original Greek *paradeigma*, i.e., *example*. Therefore, paradigms are examples. These examples gain their value, which is ultimately a practical value ("to provide models"), from offering a diverse body of information ("law, theory, application, instrumentation"). As such, Kuhn's definition applies well also to engineering, particularly to the design of hardware and software systems in computer engineering.

In their work, engineers naturally follow fundamental laws and theories, but, as they strive to build their systems on time, they regularly find support in those practices, methods, and tools which have been applied repeatedly and successfully before. And they continue to do so as long as the shared paradigm remains effective for solving their engineering

Manuscript received April 28, 2015; revised August 16, 2015; accepted September 6, 2015. Date of publication October 15, 2015; date of current version October 26, 2015. This work was supported in part by the National Science Foundation and by C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

The author is with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: luca@cs.columbia.edu).

Digital Object Identifier: 10.1109/JPROC.2015.2480849

problems. Conversely, the emergence of a growing number of anomalies that are intractable with the established paradigm leads a community of scientists and engineers to start searching for a paradigm change. Eventually, after a period of both resistance to change and proliferation of competing approaches, a new paradigm emerges [1].

This paper is about the crisis of the paradigm of synchrony in the design of integrated circuits and the emergence of a new candidate for a paradigm that looks ahead at the shift toward the engineering of systems on chips (SoCs). The crisis started around the turn of the century with the anomalies caused by global wire delays (the Wire Problem) [2]. It continues to date with the resistance to change those practices and methods that have been so successful for realizing chips with tens of millions of transistors but are ineffective for future billion-transistor SoC platforms. Latency-insensitive design (LID) is a methodology that was originally developed to address the Wire Problem, while preserving as much as possible the advantages of the synchronous assumption in register-transfer level (RTL) design. As I revisit LID and related works, I show how its principles are informing the transition from RTL design to system-level design (SLD).

II. THE SYNCHRONOUS PARADIGM

The synchronous design paradigm, or simply synchronous paradigm, is ubiquitous in electrical engineering and computer science [3]. It is the basis of digital integrated circuit design, it is used in building discrete-time dynamical control systems, and it is the foundation of programming languages and design environments for real-time embedded systems [4]. With the synchronous paradigm, a complex system is represented as a collection of interacting concurrent components whose state is updated collectively in one instantaneous step. The system consists of a composition of sequential functional processes and evolves through a sequence of atomic reactions. At each reaction all processes, simultaneously, read the values of their input variables and use them, together with the values of their state variables, to compute new values for both their state and output variables. Between two successive reactions the communication of the computed values across the processes occurs via broadcasting.

The synchronous hypothesis is precisely the idea that at each reaction the computation phase (within the functional modules) and the communication phase (transferring the computed values across modules) occur in sequence without any overlap between them. Each of the two phases can be thought as instantaneous with respect to the other. Indeed, in the synchronous paradigm, "time" progresses in lock step, one reaction after the other. Measuring time is confined to the concept of a virtual, or logical, clock, whose ticking indexes the totally ordered sequence of reactions. Each index is denoted as a timestamp. The set of timestamps coincides with the set of natural numbers.

The power of the synchronous paradigm lies essentially in its simplicity. It is an intuitive, but formal, model of computation that offers many advantages as it: 1) simplifies the modeling of deterministic concurrent systems; 2) enables the incremental design of complex systems in a modular and hierarchical fashion; 3) facilitates the design process by separating functionality from the notion of time; 4) encourages abstraction and reuse by leading to design specifications that are independent from the details of a particular implementation technology; and 5) eases the development of design automation tools for specification, validation, and synthesis.

In summary, to use the words of Benveniste *et al.*, "the paradigm of synchrony has emerged as an engineer-friendly design method based on mathematically sound tools" [3].

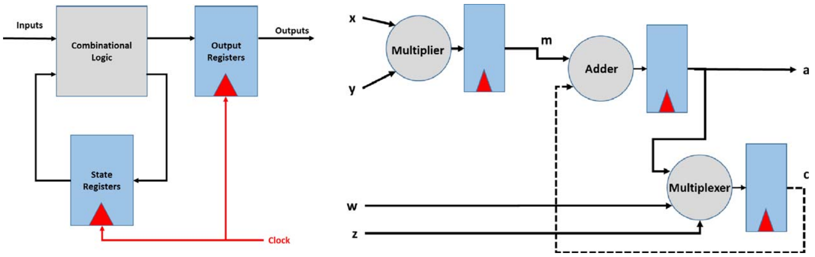
A. Synchronous Paradigm and Hardware Design

In digital hardware design, methodologies and tools based on the synchronous paradigm have made it possible to turn the progress of Moore's Law [5] into generations of integrated circuits (IC), each generation more powerful and more complex. To build these ICs, engineers assemble a myriad of transistors whose concurrent operations are tightly controlled by the beat of a master clock signal (the physical, or real, clock). Transistors and logic gates, however, are abstracted away during most stages of the design process. Since the mid-1980s, the core of the design effort occurs at the RTL of abstraction, where designers use hardware-description languages (HDLs), like Verilog or VHDL, to write circuit specifications. These specifications are processed by computer-aided design (CAD) tools, which automatically generate the circuit implementations through logic and physical synthesis [6].

Fig. 1(a) shows the block diagram of a sequential module, i.e., the basic RTL building block for applying the synchronous paradigm to IC design. The acyclic combinational logic implements the functionality of the module (typically a complex arithmetic or logic function) while the registers (memory elements controlled by the clock) store the values of the state and output variables over time. A sequential module is the direct implementation of a finite state machine (FSM),[1] which is the model of computation used to specify control logic in IC design. Also, most arithmetic data paths can be modeled as a cascade of sequential modules. Using HDLs, designers write software programs to specify the functionality of each module and their (possibly hierarchical) composition.

Example: A multiplier accumulator (MAC) is a common digital circuit used to implement operations like $\sum_k x(n) \cdot y(n-k)$, which are ubiquitous in signal processing.

[1] Specifically, the diagram of Fig. 1(a) shows the basic implementation of a Moore FSM. Removing the output registers yields that of a Mealy FSM [7].



(a) (b)

	Input				*Internal*		*Output*
n	x	y	w	z	m	c	a
1	2	2	0	0	0	0	0
2	1	3	1	0	4	0	0
3	5	4	2	0	3	0	4
4	8	1	3	0	20	4	3
5	4	5	4	0	8	3	24
6	6	6	10	1	20	24	11
7	7	7	5	0	36	10	44
8	3	8	6	0	49	44	46
9	2	2	7	0	24	46	93
10	–	–	–	–	4	93	70
11	–	–	–	–	–	–	97

(c)

Fig. 1. ***(a) RTL diagram of a generic sequential module. (b) RTL block diagram of the MAC circuit. (c) Example of RTL traces for the MAC circuit.***

Fig. 1(b) shows the RTL diagram of a pipelined implementation of a variation of a MAC circuit, which returns alternate accumulated sums: x and y are the input values to be multiplied, c is the output of the accumulator register that can be preset to a given value w by setting input $z = 1$, and a is the resulting sequence of alternate partial sums. The circuit is designed as the composition of three sequential modules. Its RTL behavior is captured by three equations with $n \in \mathrm{IN}$ denoting the timestamp

$$\begin{aligned} m_{n+1} &= x_n \cdot y_n \\ a_{n+1} &= c_n + m_n \\ c_{n+1} &= \begin{cases} a_n, & \text{if } z_n = 0 \\ w_n, & \text{if } z_n = 1. \end{cases} \end{aligned}$$

The table of Fig. 1(c) reports the traces for a fragment of an RTL behavior spanning 11 clock periods. The circuit performance is dictated by the clock frequency sustainable by its slowest component, which in this case is likely to be the multiplier. ■

The key point in RTL design is the separation of the design and validation of the functional behavior of the system from the analysis and optimization of its performance. The longest combinational path inside a module (critical path) dictates the minimum clock period ψ_i that makes it operate correctly. Therefore, given a target clock period ψ for the clock signal, the task of designing a large digital circuit can be decomposed in subtasks aimed at designing smaller RTL modules such that $\psi_i < \psi$ for each module i. The modules can be specified, simulated, implemented, and verified independently from each other, based only on the desired input/output (I/O) functionality and the expected value of ψ. The separation of functional design from performance analysis consists of the following: a functionally correct design is obtained by simply assembling all the modules, while its speed (i.e., clock frequency) is determined by the slowest module. In other words, once all modules are composed, the overall circuit works correctly as far as it is running with a clock having a period $\psi \geq \max_i\{\psi_i\}$. The effectiveness of this strategy lies on the assumption that the delay of any path connecting two modules (intermodule delay) is negligible compared to the delay of the paths inside the slowest module in the system (intramodule delay). This had been the case for the first 30 years of progress of semiconductor technologies, when the average intermodule delays were much smaller than the delays of the logic gates.

The other advantages of the synchronous paradigm also apply to hardware design. In particular: 1) the design of the circuit as a deterministic concurrent system is simplified (different modules of the same system can be designed independently by different designers); 2) the RTL design is inherently incremental and hierarchical (e.g., portions of the circuit can be redesigned to improve their performance without touching the rest of the system); 3) the RTL design of a module is independent from the particular semiconductor technology used to build the circuits; 4) predesigned RTL modules can be reused for different projects; and 5) designers can take advantage of a rich offering of commercial CAD tools for RTL specification, simulation, synthesis, and validation.

B. Crisis of the Synchronous Paradigm for IC Design

The crisis of a paradigm begins with its blurring in the attempt of dealing with a growing set of anomalies [1].

While strongly simplifying system specification, the synchronous hypothesis leaves the problem of deriving a correct and efficient implementation from it. The difficulty of this problem grows dramatically when the physical implementation has a distributed nature that poorly matches the synchronous hypothesis. This has been increasingly the case for IC design since the turn of the century, with the progress of semiconductor manufacturing toward nanometer technologies [8]. These enable a denser integration of transistors and modules on a chip. However, while local wires connecting devices within a module shrink with its logic, global wires connecting devices across different modules do not because they need to span significant parts of the die [2]. The increasing resistance–capacitance delays combined with the growth of clock frequencies, die size, and average interconnect length cause many intermodule paths to become critical paths. As the percentage of the die reachable within a clock period decreases [9], more and more gates can fit on a chip than can communicate in a clock period [2]. The Wire Problem is exacerbated by the difficulty of estimating global interconnect delays early in the design process [10]. In turn, since every violation of the target clock period is a design exception that needs to be fixed, designers are forced to many costly iterations across the various stages of the CAD flow before converging to a final implementation (timing closure problem) [11]. In summary, the chip has become a distributed system, proliferating wire-related design exceptions act as anomalies for the synchronous paradigm, and projects shift from being computation bound to being communication bound [10].

Hence, the crisis of the synchronous paradigm starts as the consequence of a spreading gap between the synchronous hypothesis of the specification and the distributed reality of the implementation. On the one hand, to assume instantaneous communication when it is more likely that the concurrent modules will be implemented as distributed components may lead to poor design specifications. On the other hand, even if it is still possible to take a synchronous specification and enforce a synchronous-design style on the distributed implementation, the final result is likely to be a suboptimal design.

But has this crisis really started? After all, the vast majority of today's digital ICs are still designed starting from RTL specifications as synchronous circuits and implemented as chips controlled with a master clock signal. Still, it is a fact that the design of high-performance ICs is becoming increasingly more difficult and expensive [12], [13].

While it may be debatable whether the crisis of the synchronous paradigm started, it is clear that, if it has, it has not ended. The proof is that a new candidate paradigm has not emerged yet. LID was proposed as an attempt to end this apparent crisis with a compromise: preserve the synchronous hypothesis while relaxing the time constraints during the early phases of the design process when correct measures of the delay paths among the modules are not yet available. Being a compromise, it could have developed in either of two directions: as a confirmation of the synchronous paradigm or as a candidate for a new paradigm. About 15 years since it was first proposed [14], the odds, I believe, are increasingly in favor of the second outcome. I explain my rationale in Section V, after offering a retrospective on LID.

III. LATENCY-INSENSITIVE DESIGN

The theory of LID is the foundation of a correct-by-construction methodology to design complex systems by assembling predesigned components. LIDs are synchronous, distributed systems and are built by composing functional modules that exchange data on communication channels according to a latency-insensitive protocol. The protocol works on the assumption that the modules are stallable, a weak condition to ask them to obey (as explained below). The goal of the protocol is to guarantee that a system composed of functionally correct modules behaves correctly independently of the channel latencies. This increases the robustness of a design implementation because any delay variations of a channel can be "recovered" by changing its latency while the overall system functionality remains unaffected. As a consequence, an important application of the proposed theory is the LID methodology to build SoCs with nanometer technologies [15].

A. The Protocols and Shells Paradigm

Latency-insensitive protocols are a mechanism to formally separate communication from computation by specifying a system as a collection of computational processes that exchange data through communication channels. The protocols cause the communication to be insensitive to the latencies of the channels. The theory of LID can then be applied as a rigorous basis to design complex systems by simply composing predesigned and verified components so that the composition satisfies, formally and by construction, the required properties of synchronization and communication. Also, the theory naturally enables the separation of the system specification from the derivation of one among many possible implementations. The designers of a latency-insensitive system can focus first on specifying the overall system and then on choosing the best components for the implementation. In doing so they do not need to worry about communication details such as data synchronization and transmission latency. A latency-insensitive protocol takes care of these. Further, as the designers explore the design space and consider alternative implementations for the various parts of the system, they can rely on the fact that the implementation of the interface (i.e., the shell) between the component and the latency-insensitive channel can be automatically generated.

A formal presentation of the theory of LID was given on the basis of the "tagged-signal model" denotation framework [16]. Section III-B and III-C summarize the main concepts of LID and the methods for their practical application, respectively.

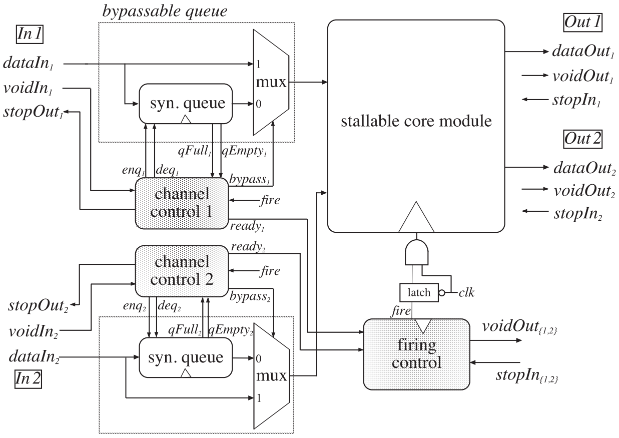
B. Theory of LID

At the core of LID lies the notion of latency equivalence: two signals are latency equivalent if they present the same ordered streams of data items but possibly with different timing. In a synchronous model of computation the existence of a clock guarantees a common time reference among signals and, therefore, a signal must present an event at each clock period [3], [17]. LID distinguishes between the occurrence of an informative event (a valid data item also known as a valid token or a true packet) and a stalling event (a void token or a void packet, often denoted with the τ symbol). Any class of latency-equivalent signals contains a single reference signal that does not present stalling events (a strict signal) while all the other members of the equivalence class (stalling signals) contain the same sequence of informative events interleaved by one or more stalling events. By following the tagged-signal model [17], the notions of latency-equivalence signals, strict signals, and stalling signals are extended to behaviors (i.e., sets of signals) and processes (i.e., sets of behaviors).

A synchronous system can be modeled as a set of processes communicating by exchanging signals (sequences of events) on a set of point-to-point channels. In the theory of LID, each signal is associated to a distinct channel. Two systems are latency equivalent if on every channel they present the same data streams, i.e., the same ordered sequence of data items, but, possibly, with different timing. The sets of signals of two latency-equivalent systems are latency equivalent. In particular, when solicited by latency-equivalent sequences of events occurring at their input ports, two latency-equivalent systems produce latency-equivalent sequences of events at their output ports.

As discussed in Section II-B, the Wire Problem causes delay uncertainties in intermodule (i.e., interprocess) communication. This greatly complicates the use of strict equality to compare corresponding signals across the various stages of the design process, from the initial RTL specification to the final layout implementation. Once one must abandon strict equality, latency equivalence becomes the next most natural concept to build upon. The first goal of LID is to accommodate this shift within the realm of the paradigm of synchrony while keeping things as simple as possible. Specifically, LID expects that designers continue to build a complex synchronous system as if it only contained strict signals which can be compared in terms of strict equality. From the strictly synchronous specification of a system, then, designers can use CAD tools to obtain automatically a particular implementation that is latency equivalent to it. This automatic transformation is made possible by the main theoretical contributions of the theory of LID. These are about the concept of "patience."

1) Patience: A patient system is a synchronous system whose functionality depends only on the order of the events of each signal and not on their exact timing. A latency-insensitive protocol guarantees that a patient system, if composed of functionally correct processes, behaves correctly independently from the delays of the channels connecting the processes. The compositional nature of the concepts of patience and latency equivalence is proven with the following three results [16]:[2] 1) the intersection of two patient processes is a patient process; 2) given two pairs of latency-equivalent patient processes, their pairwise intersections are latency equivalent; 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 pairwise intersections are latency equivalent.



$$
\begin{aligned}
\mathit{fire} &= \bigwedge\nolimits_{i\in\mathcal{I}}(\overline{voidIn_i} + \overline{empty_i}) \cdot \overline{\bigvee\nolimits_{j\in\mathcal{O}}(stopIn_j \cdot \overline{voidOut_j})} \\
\forall j \in \mathcal{O}\ voidOut_j{}^{+} &= \begin{cases} 0 & \text{if } stopIn_j \cdot \overline{voidOut_j} \text{ is true} \\ \overline{\mathit{fire}} & \text{otherwise} \end{cases} \\
\forall i \in \mathcal{I}\ stopOut_i &= full_i \\
\forall i \in \mathcal{I}\ enq_i &= \overline{voidIn_i} \cdot (\overline{\mathit{fire}} + \overline{empty_i}) \cdot \overline{full_i} \\
\forall i \in \mathcal{I}\ deq_i &= \overline{empty_i} \cdot \mathit{fire} \\
\forall i \in \mathcal{I}\ bypass_i &= empty_i
\end{aligned}
$$

Fig. 2. ***(Top) Block diagram template for a two-input–two-output shell encapsulating a stallable core module. (Bottom) Logic functions of the shell controller.***

The major result of the theory naturally follows: if all processes in a strict system are replaced by corresponding latency-equivalent patient processes, then the resulting system is patient and latency equivalent to the original one. This result provides a formal way to orthogonalize computation and communication in SLD [18], as explained next.

2) Shell Encapsulation: The next question is: How to derive the patient processes? The answer is a procedure to transform any given strict process into a patient process that is latency equivalent to it. To do so the strict process must be stallable, i.e., it can be forced to wait for any number of clock periods without losing its internal state.[3]

The procedure consists in encapsulating the strict stallable process, which is referred as core or pearl in LID, with a special process called shell, as shown in Fig. 2. This

[2] These results use the parallel composition by intersection of processes, a method that is typical of the synchronous paradigm.

[3] Stallability is a requirement that is much easier to satisfy than patience. For instance, at the specification level, any sequential module can be modeled as an FSM and, therefore, can easily be made stallable by adding an extra Boolean input variable and extra Boolean output variable to denote stalling input and output events, respectively: when active, this input variable forces the FSM to remain in the current state and to raise this output variable at the next clock period. At the implementation level, stallability can be implemented through clock gating, a pervasive technique for low-power design [19].

TABLE 1 RTL Traces for the LID Version of the MAC Circuit of Fig. 1(b)

	Input				*Internal*		*Output*
n	x	y	w	z	m	c	a
	$d(v, s)$	$d(v, s)$	$d(v, s)$	$d(v, s)$	$d(v, s)$	$d(v, s)$	$d(v, s)$
1	2(0, 0)	2(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)	0(0, 0)
2	9(1, 0)	3(0, 0)	1(0, 0)	0(0, 0)	4(0, 0)	0(0, 0)	0(0, 0)
3	1(0, 0)	4(0, 0)	2(0, 0)	0(0, 0)	4(1, 0)	0(0, 0)	4(0, 0)
4	5(0, 0)	1(0, 0)	3(0, 0)	0(0, 0)	3(0, 0)	4(0, 0)	4(1, 0)
5	8(0, 0)	5(0, 0)	4(0, 0)	0(0, 0)	20(0, 0)	4(1, 0)	3(0, 1)
6	4(0, 0)	6(0, 0)	10(0, 0)	1(0, 0)	8(0, 0)	3(0, 0)	3(0, 1)
7	6(0, 0)	7(0, 0)	5(0, 1)	0(0, 1)	20(0, 1)	3(1, 1)	3(0, 0)
8	7(0, 0)	8(0, 1)	5(0, 1)	0(0, 1)	20(0, 0)	3(1, 0)	24(0, 0)
9	3(0, 0)	8(0, 0)	5(0, 0)	0(0, 0)	36(0, 0)	24(0, 0)	11(0, 0)
10	2(0, 0)	2(0, 0)	6(0, 0)	0(0, 0)	49(0, 0)	10(0, 0)	44(0, 0)
11	—	—	7(0, 0)	0(0, 0)	24(0, 0)	44(0, 0)	46(0, 0)
12	—	—	—	—	4(0, 0)	46(0, 0)	93(0, 0)
13	—	—	—	—	—	93(0, 0)	70(0, 0)
14	—	—	—	—	—	—	97(0, 0)

encapsulation returns a patient process that is latency equivalent to the original strict process. As a circuit, the shell implements a wrapper module that uses buffering queues and synchronization logic to act as an interface between its core and the latency-insensitive protocol governing the channels. At every clock period, the shell decides between two main actions: if a new set of valid tokens has arrived through the input channels, then the shell lets the core consume them to produce new valid tokens that are transmitted on the output channels at the next clock period; if, instead, there is an input channel that does not have a new valid token, then the shell stalls the core while storing in its queues those valid tokens that may have arrived on other input channels (so that they will be processed in a future clock period).

Fig. 2 shows a particular shell implementation among many possible ones; in fact, one may choose among many different latency-insensitive protocols [20], [21]. There are, however, some common properties. In particular, backpressure is a mechanism that lets a downlink shell on a given channel request the uplink shell to temporarily stop its production of valid tokens. The reason for this request is that the buffering capability of a shell in terms of input queues is finite. Hence, the cause of many consecutive stalling periods is either the sustained lack of alignment of corresponding valid tokens across input channels or backpressure coming from output channels (or, possibly, their combination) [22]. In the case of Fig. 2, the latency-insensitive protocol that governs each channel uses two signals: a void bit to distinguish void from valid tokens and a stop bit to implement backpressure. The control logic is general and can be easily scaled to handle an arbitrary number of input and output channels. The clock-gating signal (*fire*) decides whether the core module is fired or stalled. It is asserted when each channel presents a valid token either directly from the input channel or from its input queue, and no stop request has arrived on any output channel. The second condition can be detected by checking the current *stopIn* and *voidOut* bits for each output channel. The condition $\mathit{stopIn\_j} \cdot \overline{\mathit{voidOut\_j}} = \mathit{true}$ means that the downlink module on channel *j* was not able to process the current (also latest) valid data token. If so, the core module will be stalled, the current token will be repeated, and *voidOut_j* will be set low. In all other cases, the value of the *voidOut_j* bit depends on whether the core module will be fired. The major data path components in a shell are the bypassable queues that store unused valid tokens from input channels. If a queue is empty, the shell controller selects the data token from its corresponding input channel and passes it to the core.

Example: Table 1 reports the traces for the RTL behavior of a LID of the MAC circuit of Fig. 1(b). This design is obtained by encapsulating each of the three sequential modules with a corresponding shell automatically generated from the template of Fig. 2. The columns for each signal in Table 1 show the values for three variables (d, v, s), which are abbreviations for the data, void, and stop signals of each intermodule channel. For example, at timestamp $n = 1$, the input channel x receives valid data (equal to 2) from the environment, while at $n = 2$, the data (equal to 9) are not valid because $\mathit{voidIn}_x = 1$. The arrival of these invalid data causes a stalling period for the multiplier at $n = 2$ and, consequently, channel m keeps the same value (4) at $n = 3$ but labels it with $\mathit{voidIn}_m = 1$ to avoid double sampling from the downlink adder. In turn, this causes a stalling period for the adder at $n = 3$ (the repeated 4 on output channel a is marked as void at $n = 4$) and for the multiplexer at $n = 4$ (and, therefore, channel c has a void value at $n = 5$). At both timestamps $n = 5$ and $n = 6$, the environment requires output channel a to keep the same value (3) by raising $\mathit{stopIn}_a = 1$; the environment samples this value only at $n = 7$. Meanwhile, this request has caused the adder to stall twice and the multiplier and multiplexer to stall once. The stalling requests propagate back to the input channels w and z

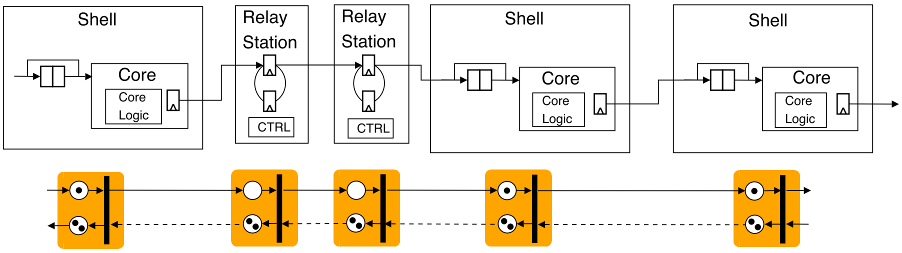


Fig. 3. ***(Top) A fragment of a system with three shells and two relay stations, which pipeline a channel; void and stop signals are omitted to avoid cluttering. (Bottom) The marked-graph model corresponding to the fragment: each component has a single corresponding transition in the marked graph. The tokens in the places denote the number of valid data items (in the forward direction) and the number of available buffering slots (in the backward direction).***

appearing at $n = 7, 8$; one stalling request appears also on input channel y ($stopIn_y = 1$ at $n = 8$), while the other is absorbed by the multiplier shell, which uses the opportunity to realign the occupations of its queues for x and y that were misaligned since the arrival of the void value on channel x at $n = 2$. By comparing each column of the table of Fig. 1(c) with the corresponding triplet of columns of Table 1 it is easy to check that the data signals on the corresponding channels in the two systems are latency equivalent. In doing so, it is important to keep in mind that for this latency-insensitive protocol at any given timestamp the activation of at least one signal between the void and stop signals encodes the presence of a stalling event on the corresponding channel. ■

Typically, the shell design is optimized to minimize area or performance overhead (e.g., by making the input queues bypassable [20]). The shell can be automatically generated from just the I/O signal specification of a core; this guarantees the broad applicability of LID since no information on the internal structure/behavior of a core is required beyond stallability. If available, this information enables performance optimizations [21].

3) Channel Pipelining: The shell encapsulation procedure expands the capabilities of RTL design by making it robust with respect to the Wire Problem. For instance, a design given as a netlist of Verilog modules (or VHDL entities) can be made patient by synthesizing automatically a shell around each of them. The resulting netlist can then proceed through the steps of a standard CAD flow (logic synthesis, physical design, etc.) with a new design option, wire pipelining: those wires implementing a channel that have a delay greater than the target clock period can be segmented into shorter wires through the insertion of sequential repeaters. This is a well-known technique to trade off the fixing a wire exception with increasing its latency by one or more clock periods. As the number of wire exceptions has grown dramatically with the arrival of nanometer technology processes [2], LID simplifies wire pipelining by making it amenable to automation. Any channel between two pairs of shell–core pairs can be pipelined through the insertion of one or more sequential repeaters, called relay stations, without any need of revisiting the logic of either the cores or the shells (Fig. 3). In breaking a combinational path, relay stations act similarly to traditional clocked repeaters [23], [24], but differ because they are patient processes which implement a latency-insensitive protocol.[4]

The compositionality of latency equivalence for patient processes guarantees that from a given specification of a strict RTL design, it is possible to obtain a class of many design implementations (which are patient and latency equivalent to it). Thanks to the latency-insensitive protocol and shell encapsulation, channel pipelining with relay stations can be done without changing the logic of any process of the original design. The class of design implementations, which differ only for the number of relay stations, embodies the flexibility of LID in a modular and scalable way.

As for the shells, the relay-station implementation depends on the choice of a latency-insensitive protocol [20], [21]. Common properties include the twofold data-storage capacity and unit latency in transferring data (in the forward direction) and backpressure (in the backward direction). Fig. 4 illustrates an implementation compatible with the shell of Fig. 2. The control logic implements a two-state Mealy FSM. The initial state is the processing state, which enables the main flip-flop and sets the *stopOut* bit low. The switching from the processing state to the stalling state is triggered by the condition that the *stopIn* bit is high, and both the *voidIn* and *voidOut* bits are low. In the stalling state, the relay station uses both the main and auxiliary flip-flops to store two valid tokens while requesting the uplink sender to stop sending valid tokens by asserting its *stopOut* bit. The relay station goes back from the stalling to the processing state when its downlink receiver deasserts the *stopIn* bit, indicating that it is ready to receive new valid tokens; hence, the relay station moves the token saved in the auxiliary flip-flop to the main flip-flop.

Being a sequential module, a relay station must be initialized at startup time. Since relay stations are added to

[4] Indeed, after the insertion of some relay stations, a pipelined channel itself can be seen as a patient process that is latency equivalent to the original strict process, i.e., the channel without relay stations.

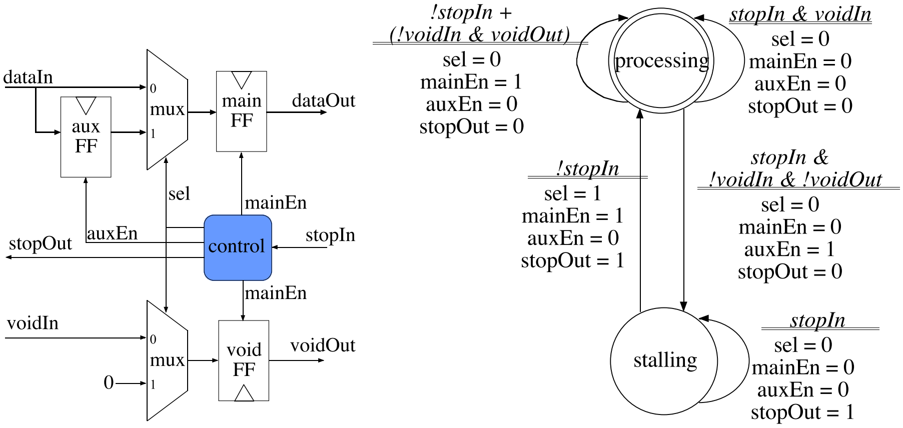


Fig. 4. ***(Left) Block diagram template for a relay station. (Right) State transition diagram of its control FSM.***

an RTL design *a posteriori* (i.e., after the design is completed) they can only be initialized with a void token. Indeed, relay stations are the origin of void tokens within[5] a latency-insensitive system, a fact of critical importance for practical purposes [22].

C. Practice of LID

The theoretical results of LID provide this formal guarantee: from any given RTL design specification made of stallable processes, it is possible to synthesize a class of many implementations which are correct by construction in the sense that their behavior is correct independently from the latency values of the interprocess communication channels. The criterion to evaluate the preservation of the correct semantics while moving from the specification to a particular implementation is, of course, the notion of latency equivalence. Any implementation is the result of choosing a particular arrangement of relay stations on the channels to satisfy the target clock period ψ (see Section II-A). This semantics-preservation guarantee is a strong theoretical result of LID. But considering that the final implementation produces not only valid tokens but also void tokens, one may wonder about its quality from a practical viewpoint.

1) Nominal versus Effective Clock Frequency: In order to evaluate the performance of a latency-insensitive system S, it is necessary to check how frequently it produces void tokens at its output ports. Accordingly, the throughput $\vartheta(S)$ of a latency-insensitive system S is defined as its rate of production of valid tokens over time. This is a number between zero and one that corresponds to the ratio of valid tokens over the sum of valid and void tokens (as observed at the system outputs) [25]. Hence, given a target clock period ψ, if S runs nominally at the clock frequency $\phi_{\text{nom}} = 1/\psi$, then effectively it produces valid data at a rate $\phi_{\text{eff}}(S) = \phi_{\text{nom}} \cdot \vartheta(S)$. Throughput $\vartheta(S)$ depends on two factors: the internal structure of S and the interaction with the environment E where S operates [22].

2) Maximum Sustainable Throughput: A latency-insensitive system S may receive void tokens at its primary inputs from the environment E in which it operates as well as generate them internally by itself. In the first case, obviously, E impacts the data-processing throughput $\vartheta(S)$ of S. The second case is more interesting from a design perspective because it sets a limit on the maximum throughput that S can sustain regardless of the characteristics of E. A properly designed shell emits void tokens on its output channels only as a result of being forced to stall. At system startup, each core, being a sequential process, has its output registers initialized with a valid token; each relay station, instead, is initialized with one void token. Since each core–shell pair operates according to an AND-causality semantics [26], the arrival of a void token on (at least) one of its input ports produces void tokens on its output ports. While some of these void tokens may leave S after a transitional phase, others may continue to loop in S forever. In the latter case, they have a negative impact on $\vartheta(S)$.

The presence and extent of throughput degradation depends on the computation structure of S, which can be formally analyzed with marked graphs [22], a restriction of the Petri Nets model of computation [27]. Fig. 3 sketches a marked graph model for a generic path in a latency-insensitive system. For any given system, the model can be unambiguously derived by combining two simple types of building blocks (one for the shell and one for the relay station) while following its topology [28]. A throughput degradation is caused by the insertion of relay stations on some particular channels (those belonging to feedback loops or reconvergent paths of system S) and can be exacerbated by backpressure effects, depending on its computation structure (its topology) [22], [28].

[5] Of course, void tokens and stop signals can also arrive at the inputs of the system when originated from the external environment.

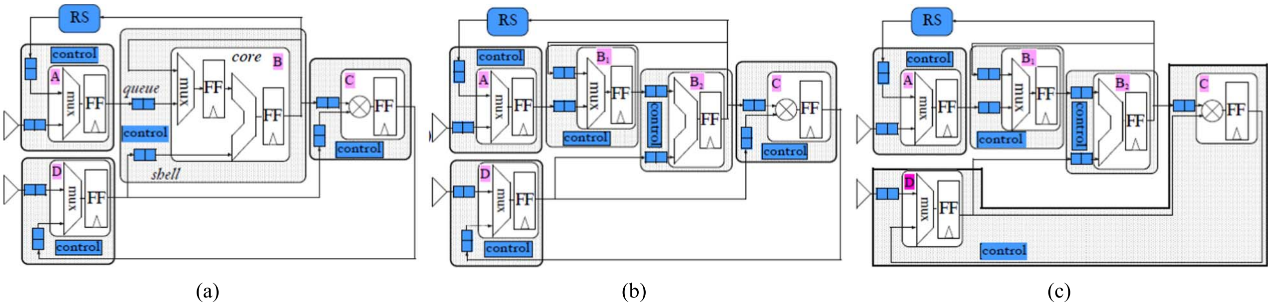


Fig. 5. *Three latency-equivalent implementations of a simple system. Different shell encapsulations lead to different MSTs: (a) 0.67; (b) 0.75; and (c) 0.75.*

The maximum sustainable throughput (MST) of a marked graph G_S modeling S is defined as [22]

$$\vartheta(G_S)=\begin{cases}1, & \text{if } G_S \text{ is acyclic}\\ \min\left\{1,\frac{1}{\pi(G_S)}\right\}, & \text{if } G_S \text{ is cyclic,}\\ & \quad\text{strongly connected}\\ \min_{\forall G_i\in\Sigma(G_S)}\{\vartheta(G_i)\}, & \text{otherwise.}\end{cases}$$

When S is modeled by an acyclic marked graph, it can sustain any rate of production/consumption regardless of the number of relay stations that are inserted on any of its channels: hence, the value of $\vartheta(G_S)$ is ideal, equal to one.

When S is modeled by a cyclic and strongly connected marked graph, then $\vartheta(G_S)$ is equal to the reciprocal of its cycle time $\pi(G_S)$, which is the average time separation between two consecutive firings of any transition of G_S (and correspondingly of any shell in S). For any cycle c of G_S, let $m(c)=s(c)/(s(c)+r(s))$ be its cycle mean, where $s(c)$ is the number of shells and $r(c)$ is the number of relay stations that are present on the cyclic path of S corresponding to c. The critical cycles of G_S are those with the smallest value of $m(c)$, which is equal to the value of $\vartheta(G_S)$.

Finally, when G_S is cyclic with a set $\Sigma(G_S)$ of strongly connected components, the value of $\vartheta(G_S)$ is effectively determined by the slowest among them.[6]

3) Optimizing Throughput by Moving Latency Around: The concept of critical cycle and the MST play the same roles for LID as the concept of critical path and clock frequency do for traditional hardware design with the synchronous paradigm (Section II-A). In assembling multiple modules to derive a system S, the module with the smallest MST constrains the MST of S in the same way as the module with the longest critical path in a synchronous circuit constrains its target clock period. LID offers an efficient solution to handle both intermodule critical paths (through automatic wire pipelining) and intermodule critical cycles. Since the effectiveness of LID depends on the ability to maintain a sufficient throughput in the presence of increased channel latencies, its application must be guided by efficient methods of performance analysis and optimization [25], [28], [30].

Example: Fig. 5(a) shows a simple latency-insensitive system S_a with four cores and one relay station; its cycle $A\rightarrow B\rightarrow RS\rightarrow A$ is critical because with two cores and one relay station it imposes the lowest bound on the MST, which is $\vartheta(S_a)=(2/(2+1))=0.67$. The throughput degradation, however, can be mitigated if the concurrency among the cores on the critical cycle is increased by a finer grained shell encapsulation of their logic. For example, Fig. 5(b) shows that the logic within core B can be partitioned into two smaller cores B_1 and B_2, each with its own shell. This partitioning yields a system S_b that is latency equivalent to S_a but has an MST $\vartheta(S_b)=(3/(3+1))=0.75$, i.e., 13% higher than $\vartheta(S_a)$. The improvement comes from the fact that in S_b cores B_1 and B_2 do not have to stall at the same clock period, while in S_a, their logic is stalled whenever the shell of B receives a void input or backpressure. Conversely, decreasing the granularity of shell encapsulation on noncritical cycles can reduce the shell's area overhead without hurting the MST. For example, since cycle $C\rightarrow D\rightarrow C$ is not critical, cores C and D can be merged so that they get encapsulated by one shell (instead of two), yielding system S_c of Fig. 5(c), which has two less queues and one less control block than system S_b of Fig. 5(b), while $\vartheta(S_c)=0.75=\vartheta(S_b)$. ■

4) Limits and Overhead of LID: Moving around latency in the system is not the same as removing it from the system. Some systems have strict performance requirements with characteristics that do not make them suitable for LID. An example is a real-time system with a component that contains a "watchdog timer" that progresses independently from the clock signal controlling the rest of the component logic. This component would not be stallable and, consequently, the system would not satisfy the

[6] Using marked graphs, it is easy to prove other key properties, e.g., a latency-insensitive system never deadlocks (it is live by construction), no matter how many void tokens continue to cycle within the system [29].

TABLE 2 Results of Floorplanning `stereo_vision`: Original Strictly Synchronous and Three LID Implementations

implementation	cell area (μm^2)				channel width	RS	RS width	floorplan area		MST
	core	shell	total	overhead	(bits)	(number)	(bits)	(μm^2)	overhead	MST
strict	7651710	0	7651710	1.00	–	–	–	8741760	1.00	1.00
a) LI: starting floorplan	7651710	225871	7877581	1.03	3124	9	378	9200440	1.05	0.83
b) LI: post-partitioning	7651710	939332	8591042	1.12	12685	19	1075	10151900	1.16	1.00
c) LI: post-merging	7651710	190812	7842522	1.03	2625	0	0	9005140	1.03	1.00

stallability condition specified in Section III-A. On the other hand, if a real-time system is made of stallable components, then LID can be applied to it and the functionality of the system will be correct. This, however, does not necessarily mean that the system will satisfy its real-time requirements, which are about performance. So, a LID implementation of the system may run with an effective throughput that is not high enough to satisfy these requirements in the same way as a strict implementation of the system may run at a clock frequency that is not sufficiently high. In both cases, careful design space exploration and optimization combined with the choice of an appropriate technology process are necessary to achieve an implementation that is functionally correct and meets the performance requirements.

In exchange for the flexibility of pipelining long channels and moving around latency, LID introduces some design overheads. The circuits implementing the latency-insensitive protocol in the relay stations and shells occupy area and dissipate power. These costs, however, are usually fairly small relatively to the overall design. For instance, Table 2 reports the results of applying LID to Stereo Vision, a SoC that measures stereo depth and consists of 16 instanced modules for a total of over half a million gates and about 200 000 flip-flops [31]. The results are reported for the original strictly synchronous design and for three latency-equivalent LID implementations [30]. Each row in the table reports on the cell area (broken down into core and shell areas), channel width, the total number and width of relay stations, the floorplan area, and the MST.[7] Row "*strict*" shows the results of floorplanning the strict design: since this is not latency insensitive, the shell area is zero and MST is one. Row "*starting floorplan*" shows the results of applying LID to the original SoC in a straightforward way, i.e., by simply encapsulating the cores of the original design: its MST is 0.83 with an area overhead of 5% compared to the strict design. Row "*postpartitioning*" shows the results after a large core on the critical cycle is partitioned into smaller cores to increase concurrency and raise the overall MST. The MST of the resulting LID implementation is improved to 1.0, the ideal value, but the fine-grained shell partitioning increases the area overhead to 12% in the cell and 16% in the floorplan area, compared to the strict design. This additional overhead, however, can be fully recouped after merging the shells of cores that are on noncritical cycles while preserving the overall MST, as shown by the results in row "*postmerging*." The final LID implementation has ideal MST and an area overhead of 3% compared to the original strict design. This is a relatively small SoC, and the overhead would be smaller for larger designs.

In summary, by introducing the new paradigm of protocols and shells, LID augments the design flexibility while preserving the decoupling of functional correctness from performance analysis of the synchronous paradigm and, therefore, its modularity.

IV. RETROSPECTIVE ON LID-RELATED RESEARCH

This section presents a discussion of works that are related to, or influenced by, LID. It is organized along five main axes.

A. LID and On-Chip Communication, Networks on Chip

The application of LID to the design of on-chip pipelined interconnects was studied by many researchers [24], [32]–[35]. Casu and Macchiarulo proposed a new implementation of the LID building blocks for the case when the computation of each core module can be scheduled statically [36] and applied it to the problem of throughput-driven floorplanning with wire pipelining [37], [38]. Boucaron *et al.* studied the static scheduling of some classes of LIDs [39]. Hassoun and Alpert used relay stations as the basic synchronization element to achieve simultaneous routing and buffer insertion in SoCs with many clock domains [40]. Chandra *et al.* proposed an interconnect design approach that is suitable to LID, as they observed that LID is a solution to "two fundamental challenges in designing interconnect channels in a system-on-chip: systems operating under different timing assumptions and long delays in communication between systems" [41].

Lahiri *et al.* were among the first to observe how LID specifications allow system designers to explore numerous alternative architectures for on-chip communication without incurring the large cost of verification at each step [42]. In a retrospective talk at the 2010 Design Automation Conference (DAC'10) [43], De Micheli identified the separation of computation and communication with LID as

[7] The designs were synthesized with a 90-nm complementary metal–oxide–semiconductor (CMOS) industrial standard cell library; all shells in the LID implementations have queues of size two.

one of the key principles that led to the birth of the idea of a network on chip (NoC) [44]–[47]. Modern bus standards, like the AMBA AXI protocol, support the pipelining of the wires through the insertion of "register stages" to help improving timing closure [48]. Hemani *et al.* established a link between the introduction of LID-enabled wire pipelining and the pipelining that switching elements naturally introduce in NoCs [44]. A variety of NoC architectures, protocols, implementations, and optimization methods have been proposed since the early 2000s [49], [50]. LID was adopted as the mode of operation for the components of ×*pipes*, a scalable, high-performance NoC architecture [51]–[53]. Latency-insensitive protocols can be used to optimize flit-buffer flow control for NoCs because both techniques rely on backpressure while they offer complementary advantages: automation of wire pipelining and simpler router design [54]. The idea of using latency-insensitive protocols in NoC design has been investigated by several other researchers in academia [55]–[62] and has found application in the industry, e.g., at STMicroelectronics [63]. Singh *et al.* discussed the application of LID to the design of NoCs with various topologies and multiple clock domains [56], [64]. Balkan *et al.* used a modified version of a relay station in their mesh-of-trees NoC for single-chip parallel processing [59], [65]. Abdelfattah *et al.* use LID to augment the existing wires and switches in a field-programmable gate array (FPGA) with an embedded NoC [66].

B. Latency-Insensitivity and Variable-Latency Circuit Design

Researchers at Intel and UPC showed how latency-insensitive systems can be used for the design of high-performance microprocessors, particularly to explore alternative microarchitectural pipelines via correct-by-construction transformations of instruction set architecture (ISA) models [67]. By being tolerant to changes in latency of computation and communication and by providing a practical method to separate timing and functionality, they enable the design of functional units that are optimized for the typical case instead of the worst case, thereby offering new design tradeoffs while simplifying also layout convergence [67], [68].

Casu *et al.* presented the first design of a latency-insensitive microprocessor (for the MIPS-R2000 ISA) with variable-latency functional units and showed how LID can uniformly support pipeline stalls caused by control and data hazards, late memory access, and variable-latency execution [69], [70]. Benini *et al.* first proposed the transformation of fixed-latency synchronous circuits into variable-latency units obtained automatically from standard fixed-latency designs to improve their average throughput [71]. More recently, the idea of accommodating variable-latency modules has been revisited for the design of speculative arithmetic units with high-level synthesis [72] and for determining at runtime the scheduling of operations in coarse-grained reconfigurable arrays [73]. Vijayaraghavan and Arvind developed a theory for modular refinement of synchronous sequential circuits using bounded dataflow networks that generalizes the LID theory and enables changing the latency of any module in a circuit, in addition to the channel latencies, without affecting its functional correctness [74]. A description of applying LID to an industrial IC design flow was given by Shand: in presenting the hardware/software codesign methodology of a product line of backend video processors for digital television, Shand explained how LID simplifies circuit composability, late pipeline changes for timing or functional fixes, and the alignment of simulation results against the software model [75].

C. LID versus Asynchronous Circuit Design and GALS

The LID methodology is reminiscent of many ideas proposed in the asynchronous-design community [76], including the inherent modularity of the macromodular systems of Clark and Molnar [77] and micropipelines of Sutherland [78]. The similarities and important differences between LID and asynchronous design are discussed in the original paper on the LID methodology [15]. More recently, Carmona *et al.* argued that, from a broader standpoint, LID can be considered as a discretization of asynchronous design and introduced "elasticity" as the property of a circuit to adapt its activity to the timing requirements of its computations, communications, and operating conditions [79]. Chelcea and Nowick proposed a collection of low-latency interface circuits for extending LID to designs with mixed-timing domains [80]. Many researchers studied LID for multiple clock domains [56], [64], [81]. In this context, Singh and Theobald showed how LID enables the extension of the notion of early evaluation from asynchronous to synchronous circuits [56], a technique investigated by many other researchers [21], [82]–[84].

The continued growth in complexity of SoC designs has led some researchers to seek a hybrid approach that combines the use of synchronous components with an asynchronous interconnection network to form a globally asynchronous locally synchronous (GALS) system. The GALS approach was first described by Seitz [85] and later formalized by Chapiro [86].[8] For modern SoC design, GALS could represent a winning compromise because: on the one hand, it accommodates the reuse of many components that are designed with well-accepted CAD flows for synchronous design and, on the other hand, it limits the use of asynchronous techniques to the design of an interconnection network, where the elimination of fixed-rate global clocking can provide a more scalable,

[8] Reportedly, a GALS approach was used first already in 1969 by Evans & Sutherland Computer Corp. in its LDS-1 commercial graphics system [87].

power-efficient, and robust solution [87]. These are also the reasons that motivated the development of industrial methodologies such as the "islands of synchronicity" [88]. A subset of GALS systems are the globally ratiochronous locally synchronous (GRLS) systems, where all clock frequencies are forced to be rationally related, i.e., they are submultiples of the same physical/virtual frequency [89], [90]. This restriction simplifies the implementation of synchronizers used to bridge rationally related clock domains and eliminates the round-trip delay penalty associated with the traditional GALS approach [90].

Both GALS and GRLS methods have commonalities with the protocols and shells paradigm of LID: each locally synchronous component is encapsulated within a circuit that, similarly to a shell, acts as an interface to an interconnection network by implementing a particular protocol. In a GALS system, the circuit is based on a mixed-timing interface and the network consists of an asynchronous circuit. Recently, however, the term "GALS" has been stretched to describe any system containing many components operating at different clock frequencies and directly connected through synchronizers, e.g., dual-clock first-in–first-out (FIFO) queues [87]. With this relaxed use, soon most ICs could be considered GALS systems because both chip multiprocessors and SoCs are increasingly based on multicore architectures and operate with aggressive power-management schemes [91]. In particular, the application of dynamic voltage and frequency scaling (DVFS) mechanisms to each core results in a system where different cores operate at different frequencies at any given time. These differences must be absorbed by the interconnection network, which can be implemented as either a synchronous or an asynchronous circuit at the physical level, as long as it supports a latency-insensitive protocol at the logical level.

D. LID and Desynchronization

Many emerging classes of embedded systems (e.g., in automotive electronics, aeronautics, and industrial automation) require the deployment of tightly interactive, concurrent processes on networked architectures. Like the designers of ICs with nanometer technologies, the designers of these embedded systems face a major issue: while helpful to deal with the complexity of deriving a concurrent specification, the synchronous paradigm matches poorly the distributed nature of the final implementation [92], [93]. The problem of desynchronization was defined to reconcile the use of synchronous programming of reactive and real-time embedded software [94] with the need of reaching a correct-by-construction modular deployment of this software on a distributed architecture [95], [96]. Benveniste was the first to study the similarities and differences between desynchronization of embedded software and LID. He concluded that "think synchronously—act asynchronously" emerges as a common paradigm for the compositional design of embedded systems [92]. The notions of endochrony and isochrony characterize those synchronous programs which can be distributed on an asynchronous architecture without losing semantic properties [96]. The relationship between these notions and the principles of LID was studied with a framework that formalizes the interplay among the concepts of event absence, event sampling, and communication latency in modeling distributed embedded systems [97]. The concept of polychrony, which denotes the capability of describing circuits and systems using the synchronous assumption together with multiple clocks, was introduced for the formal validation of the refinement of synchronous multiclocked programs into GALS architectures [98], [99]. Benveniste *et al.* proposed a protocol for the correct deployment of synchronous programs on loosely time-triggered architectures (LTTAs) [100], a weaker form of the strictly synchronous TTA proposed by Kopetz and Bauer [101]. A compositional theory of heterogeneous reactive systems based on tag systems [102] formalizes the concept of heterogeneous parallel composition and provides a model for matching a specification and an implementation that are heterogeneous: it was applied to the analysis of the correct-by-construction deployment of synchronous heterogeneous specifications onto the LTTAs in use in the aerospace industry [103]. By using a control mechanism like backpressure in LID, Tripakis *et al.* extended the communication-by-sampling method that characterizes LTTAs and, through a form of reachability analysis on marked graphs, obtained performance bounds on throughput and latency for an LTTA implementation [104]. Di Natale *et al.* adapted the LTTA model to the deployment on controller area networks (CANs) that are pervasive in automotive systems [105]. For these systems, Mangeruca *et al.* presented a method that used buffer-based intertask communication to preserve the synchronous semantics of embedded control software when deployed on single-processor or multiprocessor architectures [106].

E. Latency Insensitivity in SLD with FPGAs

FPGAs are a mainstream technology with many applications, ranging from embedded systems to hardware acceleration in data centers and high-performance computing. Thanks to the lower engineering costs and design times, an FPGA implementation represents an enticing alternative to an application-specific integrated circuit (ASIC) implementation for a growing set of applications and markets. The progress of FPGA technologies, however, has suffered for the difference in scaling between local and global interconnects, as discussed for ASICs and SoCs in Section II-B. Over five technology generations, from 130 to 28 nm, the speed of local communication spanning a relatively small amount of logic (about 40 000 elements of the largest FPGA device) has more than doubled, while global communication has not improved [107]. This speed mismatch exacerbates the timing-closure problem particularly for large designs which are increasingly the target of FPGA developers. Recently, Murray and Betz have presented an extensive study to quantify the cost and benefits of LID for FPGAs and concluded that, as design sizes continue to grow and CAD runtime for each design iteration increases, LID becomes increasingly attractive to interconnect large modules at the system level [107]. Kirischian *et al.* have used LID to design a communication infrastructure that enables the dynamic relocation of virtual hardware components to predetermined slots in the target FPGA of reconfigurable computing systems for multitask stream applications [108].

FPGAs are also extensively used in SoC design and validation because they offer a unique platform for prototyping and emulation of ICs and for development of application and system software. Complex SoC designs, however, do not typically fit on a single FPGA and must be partitioned across many. This task is time consuming to perform manually and leads to suboptimal results when left to the automatic effort of existing CAD tools. The main difficulty is to differentiate the functional behavior of the design from the cycle-by-cycle timing behavior of its RTL specification [109]. LID provides an effective solution to this problem and a path to obtain efficient multiple-FPGA implementations. By viewing a hardware design as a set of modules connected by latency-insensitive FIFO queues and breaking it at latency-insensitive boundaries, researchers at Intel and MIT showed how to automatically produce efficient implementations that span multiple FPGAs [109]. They reported the use of LID to realize highly modular design prototypes for multimedia [110] and wireless networks [111]. Also, LID enables the transparent insertion of aggressively pipelined logic into a preexisting design by using only spare resources in regions of low congestion, without the need of reconfiguring the entire FPGA [112].

The most recent advancement in this line of research is the development of the latency-insensitive environment for application programming (LEAP), an FPGA operating system built around latency-insensitive communications channels [113]. Targeting multiple FPGA platforms, LEAP organizes the memory hierarchy as a network of latency-insensitive channels and communicates with software by stretching channels across chip boundaries. By providing a rich set of LID abstraction layers and strong compiler support for automating implementation decisions, LEAP addresses two major issues that limit the adoption of FPGA technologies: it reduces the burden of programming an FPGA and enhances design portability, while consuming as little as 3% of FPGA area.[9] An open-source project, LEAP shows a promising path toward the SLD and programming of many emerging classes of heterogeneous computing platforms.

V. FROM RTL TO SLD THROUGH LID

A state-of-the-art SoC contains over a billion transistors that implement a variety of heterogeneous components, including many processors, specialized accelerators, memory subsystems, analog circuits, and interconnects. Besides heterogeneity, SoC designers face tighter power-density budgets and a rising impact of software on the design and validation process. The progress of CAD tools has not really kept up with the growing complexity of SoC design. Nearly 30 years after the adoption of logic synthesis and place-and-route tools, the IC industry has reached a point where it needs a new set of CAD tools that must allow a larger group of application experts to participate in creating the efficient hardware/software systems that they require [13]. This goal, along with the reduction of design costs and deployment times, demands the adoption of SLD methods that enable raising the level of abstraction when designing ICs [12].

A. Benefits of SLD

SLD is the next level of abstraction above RTL that is expected to provide a quantum leap in design productivity of complex SoCs. With SLD, the heterogeneous components of a SoC and their interactions are specified using a high-level programming language such as C [114], SystemC [115], BlueSpec [116] or MATLAB [117]. The benefits are multiple.

First, while RTL design specification with Verilog or VHDL is error prone and time consuming, SLD allows engineers to abstract away many low-level logic details and focus instead on the relationships between the data structures and the operations that characterize a given algorithm.

Second, both hardware and software engineers can simulate the SLD specification of a component as part of the whole system by using a virtual platform. Differently from the slow cycle-accurate RTL simulators, virtual platforms allow fast execution of complex application scenarios on top of the actual software stack that will be deployed with the SoC, including the operating system [118]. This permits designers to develop the SoC architecture based on the target applications that it must support. Also, a virtual platform reduces the gap between application software development and circuit hardware design by providing a framework for collaboration: programmers can run and refine their software on the hardware model, while designers can test and optimize their hardware guided also by the inputs from the programmers.

Third, RTL design optimization is based on running logic synthesis tools, which are very slow for large components and offer only a limited number of configuration knobs to explore alternative implementations. In

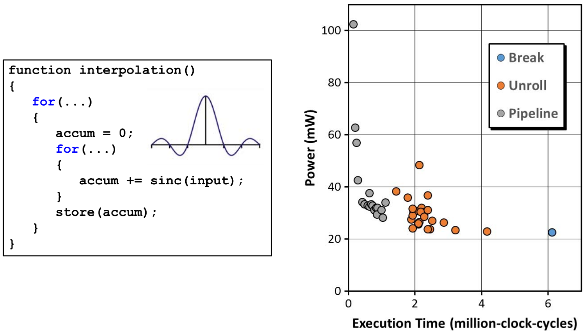
[9] This 3% overhead is consistent with the area overhead of LID in ASIC implementations, e.g., see the results of Table II.

contrast, designers can use high-level synthesis (HLS) to automatically generate many RTL implementations from a single SLD specification. Admittedly, state-of-the-art HLS tools can process only a subset of the programs that can be written with a high-level programming language [119]. But their rich set of configuration knobs allows the synthesis of many alternative microarchitectures, which are increasingly competitive with those that can be manually designed by hardware engineers. Hence, from the same SLD specification, it is possible to explore a broader design space in search of an implementation that is Pareto optimal with respect to the multiple design objectives (performance, power, area, etc.) [120]. To do the same with manual RTL design and logic synthesis would be prohibitive in terms of nonrecurring engineering (NRE) costs.

B. High-Level Synthesis and Latency Equivalence

The SLD specification of the SoC components consists of many functions (e.g., SystemC processes) that work on high-level data structures (e.g., arrays and matrices). It is the result of a sequence of partitioning and refinement steps that designers perform starting from a higher level algorithmic description. For the synthesis and optimization of an individual SystemC process, HLS tools offer a rich set of configuration knobs, e.g., for loop manipulation, state insertion, array implementation, and function sharing. The designer can choose a particular knob configuration before invoking the HLS engine, which returns a corresponding optimized microarchitecture expressed in synthesizable Verilog. Different configurations result in different microarchitectures, thus enabling the choice among many alternative RTL implementations. As these implementations represent alternative tradeoffs in the multiobjective design space, HLS promotes design reuse and intellectual property (IP) exchanges. For instance, a team of computer vision experts can devise an innovative algorithm for object detection, use SystemC to design a specialized accelerator for this algorithm, and license it as a synthesizable IP module to many different SoC designers; these can then use HLS to derive automatically the particular implementation that provides the best tradeoff (e.g., high performance or low power) for their particular system.

Example: Fig. 6 shows an example of design space exploration for an interpolation process whose main loop invokes repeatedly the *sinc* function, which is relatively expensive to realize in hardware. The diagram shows over 40 points, each corresponding to a distinct microarchitecture synthesized with HLS. The application of the "loop pipelining" knob leads to RTL implementations of the interpolation module which have more parallel and faster hardware, thereby delivering lower execution time in exchange of higher area occupation and power dissipation. Conversely, "loop breaking" leads to more sequential executions on shared hardware, resulting in resource savings but lower performance. The "loop unrolling" knob yields implementations in between those yielded by the previous two knobs. Each implementation i of the interpolation module takes a different number $p(i)$ of clock periods to execute the interpolation task on a given input. The reciprocal of $p(i)$ is equal to the MST of implementation i (see Section III-B).[10] ■



```
function interpolation()
{
   for(...)
   {
      accum = 0;
      for(...)
      {
         accum += sinc(input);
      }
      store(accum);
   }
}
```

Fig. 6. ***HLS-enabled design space exploration of an interpolation process.***

All the HLS-synthesized implementations are not functionally equivalent from an RTL viewpoint because they do not produce exactly, i.e., clock by clock, the same sequence of output signals for any valid sequence of input signals [121]. On the other hand, they are expected to be all valid RTL implementations of the original SLD specification, given as an untimed SystemC model. So, the first question to pose is: How to verify the correctness of each of these RTL implementations against the SLD specification? The answer is provided by the notion of latency equivalence. As the implementations that can be obtained with LID from a strict RTL specification are all latency equivalent to it (Section III-B), similarly all the RTL implementations that can be obtained from an untimed specification through HLS must belong to a latency-equivalent class. Every behavior of each member of this class is latency equivalent to a corresponding behavior of the untimed specification, i.e., it presents the same ordered streams of valid data items at its ports, but possibly with different timing. Hence, latency equivalence provides a key to address one of the most important challenges in the area of formal verification for SLD.

The second question is: How to choose a particular implementation among all the synthesized RTL ones? If the design consists of only one module, then it would be sufficient to analyze the set of the RTL implementations returned by the HLS tool in the multiobjective design space and choose the one that provides the desired

[10] In this example, all implementations can run at the target clock frequency (1 GHz). If this is not the case, then the effective latency, which is defined as the reciprocal of the effective clock frequency (Section III-B), should be used as the metric for the x-axis of Fig. 6.

compromise between performance and cost (e.g., for the interpolation example, choose a Pareto-optimal point in the diagram of Fig. 6). A modern SoC design, however, is the result of composing a large number of modules; further, many of these modules are themselves too complex to be synthesized by state-of-the-art HLS tools without being first broken down into smaller submodules. Indeed, SoC design is inherently an instance of component-based design. Hence, the choice of a particular RTL implementation for a module must be made in the context of the choices for all the other modules that are also components of the given SoC. A particular set of choices leads to a point in the multiobjective design space for the whole SoC. So, the process of deriving the diagram of Pareto-optimal points repeats itself hierarchically at the system level [120]. Every system-level point is the result of composing many component-level points, each corresponding to an RTL implementation that can run with its own effective clock frequency. Hence, the resulting composition works correctly only if the differences among the effective clock frequencies across all the components can be absorbed by the communication infrastructure that connects them. The protocols and shells paradigm is a modular approach to address this problem in the context of communication-based SLD.

C. Toward Communication-Based SLD

The composition of modules synthesized with HLS is just one aspect of the complexity of SoC design and programming. While the design of individual components is important, the most critical challenges in the realization of a SoC lie in the selection, integration, and management of many components.

Modern SoCs are increasingly based on heterogeneous multicore architectures that consist of a mix of cores, including many different types of programmable processors and special-function hardware accelerators. Each core can dynamically change its operating frequency depending on its current workload requirements (combined with other system conditions) and independently from the other cores.[11] All cores communicate among themselves and with off-chip devices (primarily the main memory DRAM) through a communication infrastructure. Traditionally, this has been implemented as a bus or a set of buses; however, as the number of cores continues to grow, buses are getting replaced by NoCs, which offer more scalability in terms of both logical and physical properties [45]. Since the communication infrastructure is a resource shared by all the cores, each core must be ready to temporarily stall its operations in case of congestion. Furthermore, the operations of many "device cores" (accelerator or peripherals) are intrinsically event based: at any given time, they get configured and invoked by software through device drivers, run for some time, and then get back into an idle state, typically after sending an interrupt signal that will itself be processed in an asynchronous fashion, on a best-effort basis.

In this context, communication plays an increasingly central role at both runtime and design time. At runtime, the communication infrastructure must scale up with the demands in terms of data transfers from a growing number of cores; also it must be capable to dynamically absorb differences in the effective clock frequency among the cores and provide backpressure signals to inform them about the needs for stalling without losing their internal state. At design time, communication is key to the correct and efficient assembly of components that are designed independently from each other.

Choosing the best implementation of each component for a given SoC and combining these implementations into an optimal system design are still manual, time-consuming tasks. To assist SoC designers in this effort, however, CAD-tool vendors have started to provide libraries of interface primitives. Based on the transaction-level modeling (TLM) approach [122], [123], these libraries offer: 1) an application programming interface to specify communication and synchronization mechanisms among computation processes at the system level; and 2) synthesizable implementations of these mechanisms that can be combined with the implementation of each process in a modular fashion [124]–[126]. These primitives follow the protocols and shells paradigm in using point-to-point channels, which are inherently latency insensitive, combined with modular socket interfaces, which can be instanced to connect the processes to the channels. With these primitives, TLM separates the implementation details of the communication and computation parts of the design and facilitates the combination of hardware and software components in virtual platforms. By absorbing the timing differences across processes, it simplifies the replacement of a particular implementation of any process with another one that may take a different number of clock periods, as it offers a different power/performance tradeoff point. By providing predesigned implementations and encapsulating low-level signals, it relieves SoC designers from the tedious task of creating a communication protocol. By decoupling the computation and communication parts, it enables a more efficient design of the communication infrastructure, whose implementation characteristics may vary as long as it supports the protocol.

As discussed in Section III-A, in RTL design, a latency-insensitive channel can be implemented by augmenting the wires carrying the data with two additional wires carrying the void/valid bit and the backpressure stop/ready bit, respectively. With SLD, the designer has still the option of specifying the LID protocol using a cycle-accurate model. Thanks to the interface-primitive libraries, however, modern HLS tools support more abstract

[11] Increasingly, together with the frequency also the voltage supply is scaled through the fine-grained application of DVFS.

specifications that do not require to describe the implementation details of the protocol and shell interfaces. These are automatically synthesized from the TLM specification by the tools, together with the computational part of the design. For instance, SystemC offers the `sc_fifo` data structure that can be used to specify a point-to-point channel between a producer and a consumer [115]. The higher level of abstraction simplifies the specification, debugging, and maintenance of the channels and the interfaces among the system components; it also enables faster simulation at the system level without losing the benefits of latency insensitivity either at this level or for the synthesized RTL implementation [127].

VI. CONCLUSIONS: A PERSPECTIVE FOR THE FUTURE

Building on the foundations of LID and the protocols and shells paradigm, we can bridge the gap between RTL design and SLD. If this is possible and if the benefits of SLD are so clear, one may wonder why most integrated circuits are still designed starting from manually written RTL specifications. While it is difficult to pinpoint a single cause, multiple issues are likely at play here. There is the natural inertia of continuing to apply best practices that have brought decades of successful products in the semiconductor industry. Many engineers who have been trained for RTL design and have acquired years of experience in using CAD flows that start from this level of abstraction may be reluctant to switch to new, relatively untested, practices. Meanwhile, the engineering divisions and teams of many semiconductor companies are organized in a way that is conducive to realize an integrated circuit with traditional CAD flows and their well-established signoff points. This may make managers more skeptical about the benefits of a major reorganization. These reluctance and skepticism are also amplified by some concrete challenges that delay the progress of SLD, including: the lack of a commonly accepted methodology, the limitations of current virtual platforms, HLS and verification tools, and the shortage of engineers trained to work at this higher level of abstraction.

Arguably, this is a chicken-and-egg problem: the lack of bigger investments in developing SLD methodologies and tools is due to a lack of demand from engineers; conversely, the lack of this demand is due to the shortcomings of current SLD methodologies and tools. I believe that academia should take the lead in breaking this vicious cycle. In most universities, the design and validation of digital circuits is still taught based on RTL specifications made with Verilog or VHDL. Furthermore, while the interplay between hardware and software becomes tighter with the design of each new generation of electronic products, traditional boundaries between disciplines prevent students from acquiring a true system perspective. For instance, the typical curriculum in electrical engineering (EE) does not cover basic concepts of operating systems and driver programming, while most computer science (CS) graduates who can develop sophisticated software applications cannot evaluate basic tradeoffs between performance and power dissipation.

Technology and commercial trends in electronic systems, however, call for a renewal of the professional figure of the computer engineer. The emphasis should move toward the ability of: mastering the hardware and software aspects of integrating heterogeneous components into a complete system, evaluating their performance in a multiobjective optimization space that includes both logical and physical properties, and designing new components that are reusable across different systems, product generations, and implementation platforms (e.g., FPGAs and standard cells). Based on my experience with developing the course "System-on-Chip Platforms" at Columbia University, I think that there is a growing demand for learning these skills among the new generations of EE and CS students.

According to Kuhn [1], when confronted by severe and prolonged anomalies, scientists may consider alternatives but "they do not renounce the paradigm that has led them into the crisis"; far from being a cumulative process, the transition from a paradigm in crisis to a new one "must occur all at once (though not necessarily in an instant) or not at all." A paradigm shift sometimes requires a new generation. ■

Acknowledgment

The author would like to thank present and past members of the System-Level Design Group at Columbia University, New York, NY, USA, in particular J. Chen, R. Collins, G. Di Guglielmo, C.-H. Li, H.-Y Liu, P. Mantovani, and M. Petracca. He would also like to thank L. Lavagno, A. Sangiovanni-Vincentelli, and Y. Watanabe for many discussions on topics related to this paper over the years. Finally, he thanks the anonymous reviewers for their help in improving the overall quality of the paper.

REFERENCES

[1] T. S. Kuhn, *The Structure of Scientific Revolutions*, 3rd ed. Chicago, IL, USA: Univ. Chicago Press, 2000.

[2] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.

[3] A. Benveniste *et al.*, "The synchronous languages twelve years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[4] T. A. Henzinger and J. Sifakis, "The discipline of embedded systems design," *IEEE Computer*, vol. 40, no. 10, pp. 32–40, Oct. 2007.

[5] G. Moore, "No exponential is forever: But (forever) can be delayed!" in *Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2003, pp. 20–23.

[6] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, no. 2, pp. 264–300, Feb. 1990.

[7] T. Villa, T. Kam, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: Logic Optimization*. Norwell, MA, USA: Kluwer, 1997.

[8] R. Bryant *et al.*, "Limitations and challenges of computer-aided design technology for CMOS VLSI," *Proc. IEEE*, vol. 89, no. 3, pp. 341–365, Mar. 2001.

[9] D. Matzke, "Will physical scalability sabotage performance gains?" in *IEEE Computer*, vol. 8, no. 9, pp. 37–39, Sep. 1997.

[10] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in SOC design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, Sep./Oct. 2002.

[11] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "On-chip communication design: Roadblocks and avenues," in *Proc. Conf. Hardware/Softw. Codesign Syst. Synthesis*, Oct. 2003, pp. 75–76.

[12] A. L. Sangiovanni-Vincentelli, "Quo vadis SLD: Reasoning about trends and challenges of system-level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[13] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *Proc. Int. Conf. Comput.-Aided Verif.*, Jul. 1999, pp. 123–133.

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

[16] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[17] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[18] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[19] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic, *Digital System Clocking: High-Performance and Low-Power Aspects*. New York, NY, USA: Wiley, 2003.

[20] C.-H. Li, R. Collins, S. Sonalkar, and L. P. Carloni, "Design, implementation, validation of a new class of interface circuits for latency-insensitive design," in *Proc. Int. Conf. Formal Methods Models Codesign*, Jun. 2007, pp. 13–22.

[21] C.-H. Li and L. P. Carloni, "Leveraging local intra-core information to increase global performance in block-based design of systems-on-chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 2, pp. 165–178, Feb. 2009.

[22] L. P. Carloni, "The role of back-pressure in implementing latency-insensitive design," *Electron. Notes Theor. Comput. Sci.*, vol. 146, no. 2, pp. 61–80, Jan. 2006.

[23] P. Saxena, N. Menezes, P. Cocchini, and D. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 4, pp. 451–462, Apr. 2004.

[24] L. Scheffer, "Methodologies and tools for pipelined on-chip interconnect," in *Proc. Int. Conf. Comput. Design*, Oct. 2002, pp. 152–157.

[25] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proc. Design Autom. Conf.*, Los Angeles, CA, USA, Jun. 2000, pp. 361–367.

[26] J. Gunawardena, "Causal automata," *Theor. Comput. Sci.*, vol. 101, no. 2, pp. 265–288, 1992.

[27] T. Murata, "Petri Nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[28] R. Collins and L. P. Carloni, "Topology-based performance analysis and optimization of latency-insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 12, pp. 2277–2290, Dec. 2008.

[29] L. P. Carloni, "Latency-insensitive design," Ph.D. dissertation, Electron. Res. Lab., Uni. California Berkeley, Berkeley, CA, USA, Aug. 2004, Memo. UCB/ERL M04/29.

[30] C.-H. Li, S. Sonalkar, and L. P. Carloni, "Exploiting local logic structures to optimize multi-core SoC floorplanning," in *Proc. Conf. Design Autom. Test Eur.*, Mar. 2010, pp. 1291–1296.

[31] A. Darabiha, J. Rose, and W. J. MacLean, "Video-rate stereo depth measurement on programmable hardware," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2003, pp. 203–210.

[32] R. Lu and C. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proc. Int. Conf. Comput.-Aided Design*, 2003, pp. 227–231.

[33] R. Lu and C. Koh, "Performance analysis of latency-insensitive systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 3, pp. 469–483, Mar. 2006.

[34] V. Nookala and S. Sapatnekar, "A method for correcting the functionality of a wire-pipelined circuit," in *Proc. Design Autom. Conf.*, Jun. 2004, pp. 574–575.

[35] V. Nookala and S. Sapatnekar, "Designing optimized pipelined global interconnects: Algorithms and methodology impact," in *Proc. Int. Symp. Circuits Syst.*, May 2005, pp. 608–611.

[36] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proc. Design Autom. Conf.*, Jun. 2004, pp. 576–581.

[37] M. R. Casu and L. Macchiarulo, "Floorplan assisted data rate enhancement through wire pipelining: A real assessment," in *Proc. Int. Symp. Phys. Design*, Apr. 2005, pp. 121–128.

[38] M. R. Casu and L. Macchiarulo, "Throughput-driven floorplanning with wire pipelining," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 5, pp. 663–675, May 2005.

[39] J. Boucaron, R. de Simone, and J.-V. Millo, "Formal methods for scheduling of latency-insensitive designs," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 1–16, Jan. 2007.

[40] S. Hassoun and C. J. Alpert, "Optimal path routing in single and multiple clock domain systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 11, pp. 1580–1588, Nov. 2003.

[41] V. Chandra, H. Schmit, A. Xu, and L. Pileggi, "A power aware system level interconnect design methodology for latency-insensitive systems," in *Proc. Int. Conf. Comput.Aided Design*, Nov. 2004, pp. 275–282.

[42] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 6, pp. 952–961, Dec. 2004.

[43] G. De Micheli, "Networks on chips: From research to products," in *Proc. Design Autom. Conf.*, Jun. 2010. [Online]. Available: http://infoscience.epfl.ch/record/158362/files/demicheli_dac_2010.pdf

[44] A. Hemani *et al.*, "Network on chip: An architecture for billion transistor era," in *Proc. IEEE NorChip Conf.*, vol. 31, Nov. 2000, pp. 1–8.

[45] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. Design Autom. Conf.*, Jun. 2001, pp. 684–689.

[46] L. Benini and G. De Micheli, "Networks on chip: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[47] M. B. Taylor *et al.*, "The Raw microprocessor: A computational fabric for software circuits and general purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.

[48] ARM Ltd. AMBA AXI and ACE Protocol Specification, 2011.

[49] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. D. E. Jerger, and Y. V. Hoskote, "Outstanding research problems in NoC design: System, microarchitecture, circuit perspectives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 1, pp. 3–21, Jan. 2009.

[50] J. Owens *et al.*, "Research challenges for on-chip interconnection networks," *IEEE Micro*, vol. 27, no. 5, pp. 96–108, Sep./Oct. 2007.

[51] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "×pipes: A latency insensitive parameterized network-on-chip architecture for multi-processor SoCs," in *Proc. Int. Conf. Comput. Design*, Oct. 2003, pp. 536–541.

[52] A. Jalabert, L. Benini, S. Murali, and G. De Micheli, "×pipes compiler: A tool for instantiating application-specific NoCs," in *Proc. Conf. Design Autom. Test Eur.*, Feb. 2004, pp. 884–889.

[53] D. Bertozzi and L. Benini, "A retrospective look at ×pipes: The exciting ride from a design experience to a design platform for nanoscale networks-on-chip," in *Proc. Int. Conf. Comput. Design*, Sep. 2012, pp. 43–44.

[54] N. Concer, M. Petracca, and L. P. Carloni, "Distributed flit-buffer flow control for networks-on-chip," in *Proc. Conf. Hardware/Softw. Codesign Syst. Synthesis*, Oct. 2008, pp. 215–220.

[55] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Efficient synthesis of networks on chip," in *Proc. Int. Conf. Comput. Design*, Oct. 2003, pp. 146–151.

[56] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proc. Conf. Design Autom. Test Eur.*, Feb. 2004, pp. 21008–21013.

[57] U. Y. Ogras and R. Marculescu, "Application-specific network-on-chip architecture customization via long-range link insertion," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2005, pp. 246–253.

[58] Z. Lu, I. Sander, and A. Jantsch, "Towards performance-oriented pattern-based refinement of synchronous models onto NOC communication," in *Proc. Euromicro Conf. Digital Syst. Design*, Aug. 2006, pp. 37–44.

[59] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin, "Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing," in *Proc. IEEE Symp. High-Performance Interconnects*, Aug. 2007, pp. 21–28.

[60] J. You, Y. Xu, H. Han, and K. S. Stevens, "Performance evaluation of elastic GALS interfaces and network fabric," *Electron. Notes Theor. Comput. Sci.*, vol. 200, no. 1, pp. 17–32, Feb. 2008.

[61] D. E. Holcomb, B. A. Brady, and S. A. Seshia, "Abstraction-based performance verification of NoCs," in *Proc. Design Autom. Conf.*, Jun. 2011, pp. 492–497.

[62] G. Michelogiannakis and W. Dally, "Elastic buffer flow control for on-chip networks," *IEEE Trans. Comput.*, vol. 62, no. 2, pp. 295–309, Feb. 2013.

[63] M. Coppola *et al.*, "OCCN: A NoC modeling framework for design exploration," *J. Syst. Architect.*, vol. 50, no. 2/3, pp. 129–163, Feb. 2004.

[64] A. Agiwal and M. Singh, "An architecture and a wrapper synthesis approach for multi-clock latency-insensitive systems," in *Proc. Int. Conf. Comput.-Aided Design*, 2005, pp. 1006–1013.

[65] A. O. Balkan, G. Qu, and U. Vishkin, "Mesh-of-Trees and alternative interconnection networks for single-chip parallelism," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 10, pp. 1419–1432, Dec. 2009.

[66] M. S. Abdelfattah, A. Bitar, and V. Betz, "Take the highway: Design for embedded NoCs on FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2015, pp. 98–107.

[67] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2008, pp. 434–441.

[68] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proc. Design Autom. Conf.*, Jul. 2006, pp. 657–662.

[69] M. R. Casu, S. Colazzo, and P. Mantovani, "Coupling latency-insensitivity with variable-latency for better than worst case design: A RISC case study," in *Proc. Great Lakes Symp. VLSI*, May 2011, pp. 163–168.

[70] M. R. Casu and P. Mantovani, "A synchronous latency-insensitive RISC for better than worst-case design," *Integr., VLSI J.*, vol. 48, no. 2/3, pp. 72–82, Jan. 2015.

[71] L. Benini, E. Macii, M. Poncino, and G. De Micheli, "Telescopic units: A new paradigm for performance optimization of VLSI designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 3, pp. 220–232, Mar. 1998.

[72] A. Del Barrio, S. Memik, M. Molina, J. Mendias, and R. Hermida, "A distributed controller for managing speculative functional units in high level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 3, pp. 350–363, Mar. 2011.

[73] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2013, pp. 171–180.

[74] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proc. Int. Conf. Formal Methods Models Codesign*, Jul. 2009, pp. 171–180.

[75] M. Shand, "A case study of hardware software co-design in a consumer ASIC," in *Proc. Int. Conf. Formal Methods Models Codesign*, Jul. 2011, pp. 145–150.

[76] C. van Berkel, M. Josephs, and S. Nowick, "Applications of asynchronous circuits," *Proc. IEEE*, vol. 87, no. 2, pp. 223–233, Feb. 1999.

[77] W. A. Clark and C. E. Molnar, "The promise of macromodular systems," in *Dig. Papers 6th Annu. IEEE Comput. Soc. Int. Conf.*, 1972, pp. 3009–3312.

[78] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989.

[79] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.

[80] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 8, pp. 857–873, Aug. 2004.

[81] A. Edman, C. Svensson, and B. Mesgarzadeh, "Synchronous latency-insensitive design for multiple clock domain," in *Proc. Int. SOC Conf.*, Sep. 2005, pp. 83–86.

[82] J. Julvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2006, pp. 448–455.

[83] M. R. Casu and L. Macchiarulo, "Adaptive latency insensitive protocols and elastic circuits with early evaluation: A comparative analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 245, pp. 35–50, Aug. 2009.

[84] D. Bufistov, J. Cortadella, M. G. Oms, J. Júlvez, and M. Kishinevsky, "Retiming and recycling for elastic systems with early evaluation," in *Proc. Design Autom. Conf.*, Jul. 2009, pp. 288–291.

[85] L. C. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds. Reading, MA, USA: Addison-Wesley, 1980.

[86] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, Oct. 1984.

[87] S. M. Nowick and M. Singh, "Asynchronous design—Part 1: Overview and recent advances," *IEEE Design Test*, vol. 32, no. 3, pp. 5–18, May/Jun. 2015.

[88] A. P. Niranjan and P. Wiscombe, "Islands of synchronicity, a design methodology for SoC design," in *Proc. Conf. Design Autom. Test Eur.*, 2004, pp. 64–69.

[89] J.-M. Chabloz and A. Hemani, "Low-latency maximal-throughput communication interfaces for rationally related clock domains," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 3, pp. 641–654, Mar. 2014.

[90] J.-M. Chabloz and A. Hemani, "Power management architectures in McNOC," in *Scalable Multi-Core Architectures*, A. Jantsch and D. Soudris, Eds. New York, NY, USA: Springer-Verlag, 2012, pp. 55–80.

[91] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. Int. Symp. Microarchitect.*, Dec. 2006, pp. 347–358.

[92] A. Benveniste, "Some synchronization issues when designing embedded systems from components," in *Proc. Int. Conf. Embedded Softw.*, Oct. 2001, pp. 32–49.

[93] P. Caspi and R. Salem, "Threshold and bounded-delay voting in critical control systems," in *Proc. Int. Symp. Formal Tech. Real-Time Fault-Tolerant Syst.*, Sep. 2000, pp. 70–81.

[94] A. Benveniste and B. Gerard, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.

[95] A. Benveniste, B. Caillaud, and P. L. Guernic, "From synchrony to asynchrony," in *Proc. Int. Conf. Concurrency Theory*, Aug. 1999, pp. 162–177.

[96] A. Benveniste, B. Caillaud, and P. L. Guernic, "Compositionality in dataflow synchronous languages: Specification & distributed code generation," *Inf. Comput.*, vol. 163, pp. 125–171, 2000.

[97] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "A framework for modeling the distributed deployment of synchronous designs," *J. Formal Methods Syst. Design*, vol. 28, no. 2, pp. 93–110, Mar. 2006.

[98] P. L. Guernic, J. P. Talpin, and J. C. L. Lann, "Polychrony for system design," *J. Circuits Syst. Comput.*, vol. 12, no. 3, pp. 261–303, Apr. 2003.

[99] J. P. Talpin, P. L. Guernic, S. K. Shukla, R. Gupta, and F. Doucet, "Formal refinement-checking in a system-level design methodology," *Fundamenta Informaticae*, vol. 62, no. 2, pp. 243–273, Jul. 2004.

[100] A. Benveniste *et al.*, "A protocol for loosely time-triggered architectures," in *Proc. Int. Conf. Embedded Softw.*, Oct. 2002, pp. 252–265.

[101] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.

[102] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli, "Composing heterogeneous reactive systems," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 4, pp. 1–36, Jul. 2008.

[103] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli, "Heterogeneous reactive systems modeling: Capturing causality and the correctness of loosely time-triggered architectures (ltta)," in *Proc. Int. Conf. Embedded Softw.*, Sep. 2004, pp. 220–229.

[104] S. Tripakis *et al.*, "Implementing synchronous models on loosely time triggered architectures," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008.

[105] M. Di Natale, Q. Zhu, A. L. Sangiovanni-Vincentelli, and S. Tripakis, "Optimized implementation of synchronous models on industrial LTTA systems," *J. Syst. Architect.*, vol. 60, no. 4, pp. 315–328, Apr. 2014.

[106] L. Mangeruca, M. Baleani, A. Ferrari, and A. Sangiovanni-Vincentelli, "Semantics-preserving design of embedded control software from synchronous models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 8, pp. 497–509, Aug. 2007.

[107] K. E. Murray and V. Betz, "Quantifying the cost and benefit of latency insensitive communication on FPGAs," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2014, pp. 223–232.

[108] L. Kirischian, V. Dumitriu, P. W. Chun, and G. Okouneva, "Mechanism of resource virtualization in RCS for multitask stream applications," *Int. J. Reconfigurable Comput.*, pp. 8:1–8:13, Feb. 2010.

[109] K. E. Fleming *et al.*, "Leveraging latency-insensitivity to ease multiple FPGA design," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2012, pp. 175–184.

[110] K. Fleming *et al.*, "H.264 decoder: A case study in multiple design points," in *Proc. Int. Conf. Formal Methods Models Codesign*, Jun. 2008, pp. 165–174.

[111] M. C. Ng *et al.*, "Airblue: A system for cross-layer wireless protocol development," in *Proc. Symp. Architect. Netw. Commun. Syst.*, Oct. 2010, pp. 4:1–4:11.

[112] E. Hung, T. Todman, and W. Luk, "Transparent insertion of latency-oblivious logic onto FPGAs," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Sep. 2014, DOI: 10.1109/FPL.2014.6927497.

[113] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA operating system," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Sep. 2014, DOI: 10.1109/FPL.2014.6927488.

[114] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1507–1522, Nov. 2006.

[115] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd ed. New York, NY, USA: Springer-Verlag, 2009.

[116] N. Dave, Arvind, and M. Pellauer, "Scheduling as rule composition," in *Proc. Int. Conf. Formal Methods Models Codesign*, May 2007, pp. 51–60.

[117] G. Venkataramani, K. Kintali, S. Prakash, and S. van Beek, "Model-based hardware design," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2013, pp. 69–73.

[118] D. Aarno and J. Engblom, *Software and System Development using Virtual Platforms*. San Francisco, CA, USA: Morgan Kaufmann, 2014.

[119] G. Martin and G. Smith, "High-level synthesis: Past, present, future," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul./Aug. 2009.

[120] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Proc. Conf. Design Autom. Test Eur.*, Mar. 2012, pp. 641–646.

[121] F. Somenzi and A. Kuehlmann, "Equivalence checking," in *Electronic Design Automation For Integrated Circuits Handbook*, L. Scheffer, L. Lavagno, and G. Martin, Eds. Boca Raton, FL, USA: CRC Press, 2006.

[122] F. Ghenassia, *Transaction-Level Modeling with SystemC*. New York, NY, USA: Springer-Verlag, 2006.

[123] B. Bailey *et al.*, *TLM-Driven Design and Verification Methodology*. Raleigh, NC, USA: Lulu Enterprises, 2010.

[124] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. New York, NY, USA: Springer-Verlag, 2008.

[125] M. Fingeroff, *High-Level Synthesis Blue Book*. Bloomington, IN, USA: Xlibris., 2010.

[126] J. Sanguinetti, M. Meredith, and S. Dart, "Transaction-accurate interface scheduling in high-level synthesis," in *Proc. ESLsyn Conf.*, 2012, pp. 31–36.

[127] T. Bollaert, "High-level synthesis walks the talk: Synthesizing a complete graphics processing application," in *Proc. Design Verif. Conf.*, 2011.

ABOUT THE AUTHOR

Luca P. Carloni (Senior Member, IEEE) received the Laurea degree (*summa cum laude*) in electrical engineering from the Università di Bologna, Bologna, Italy, in 1995 and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California Berkeley, Berkeley, CA, USA, in 1997 and 2004, respectively.

He is currently an Associate Professor with the Department of Computer Science, Columbia University, New York, NY, USA. He has authored over 100 publications and holds two patents. His current research interests include methodologies and tools for heterogeneous multicore platforms with emphasis on system-level design and design reuse, system-on-chip design, embedded software, and distributed embedded systems.

Dr. Carloni was a recipient of the Demetri Angelakos Memorial Achievement Award in 2002, the Faculty Early Career Development (CAREER) Award from the National Science Foundation in 2006, the Office Of Naval Research (ONR) Young Investigator Award in 2010, and the IEEE Council on Electronic Design Automation (CEDA) Early Career Award in 2012. He was selected as an Alfred P. Sloan Research Fellow in 2008. His 1999 paper on the latency-insensitive design methodology was selected for the Best of ICCAD, a collection of the best papers published in the first 20 years of the IEEE International Conference on Computer-Aided Design. In 2010, he served as Technical Program Co-Chair of the International Conference on Embedded Software (EMSOFT), the International Symposium on Networks-on-Chip (NOCS), and the International Conference on Formal Methods and Models for Codesign (MEMOCODE). He was the Vice General Chair (in 2012) and General Chair (in 2013) of Embedded Systems Week (ESWEEK), the premier event covering all aspects of embedded systems and software. He is a Senior Member of the Association for Computing Machinery (ACM).