Leveraging Local Intra-Core Information to Increase Global Performance in Block-Based Design of Systems-on-Chip

Cheng–Hong Li and Luca P. Carloni, *Member, IEEE,*

***Abstract*—Latency-insensitive design is a methodology for system-on-chip (SoC) design that simplifies the reuse of intellectual property cores and the implementation of the communication among them. This simplification is based on a system-level protocol that decouples the intra-core logic design from the design of the inter-core communication channels. Each core is encapsulated within a shell, a synthesized logic block that dynamically controls its operation to interface it with the rest of the SoC and to absorb any latency variations on its I/O signals. In particular, a shell stalls a core whenever new valid data are not available on the input channels or a down-link core has requested a delay in the data production on the output channels.**

We study how knowledge about the internal logic structure of a core can be applied to the design of its shell to improve the overall system-level performance by avoiding unnecessary local stalling. We introduce the notion of functional independence conditions (FIC) and present a novel circuit design of a generic shell template that can leverage FIC. We propose a procedure for the logic synthesis of a FIC-shell instance that is only based on the analysis of the intra-core logic and does not require any input from the designers. Finally, we present a comprehensive experimental analysis that shows the performance benefits and limited design overhead of the proposed technique. This includes the semi-custom design of an SoC, an ultra-wideband baseband transmitter, using a $90nm$ industrial standard cell library.

I. Introduction

Designers of systems-on-chip (SoC) for embedded applications face the difficult task of assembling and coordinating several hardware blocks under stringent time-to-market requirements. Latency-insensitive design (LID) has been proposed as a correct-by-construction design methodology for synchronous SoCs. LID provides a sound way to cope with the complexity of SoC design because:

1) it reconciles traditional and well-accepted CAD methods for semi-custom design, which are based on the *synchronous model of computation*, with the reality that chips designed with nanometer technologies are increasingly becoming distributed systems due to the impact of global communication delays [1];
2) it facilitates the reuse and assembly of pre-designed and pre-validated *intellectual property (IP) cores*, which can be either hard macros in GDSII format or soft macros, i.e. synthesizable logic blocks specified in a hardware description language like Verilog or VHDL [2], [3];
3) it helps SoC engineers to meet the required target clock frequency (achieve *timing closure*) and reduce the number of costly iterations in the design process by simplifying the automatic application of *wire pipelining*; this is a technique to fix timing violations in global interconnect that is very effective, yet challenging to apply [4].

C.-H. Li and L.P. Carloni are with Department of Computer Science, Columbia University in the City of New York, New York, NY 10027, USA ({cheli, luca}@cs.columbia.edu).

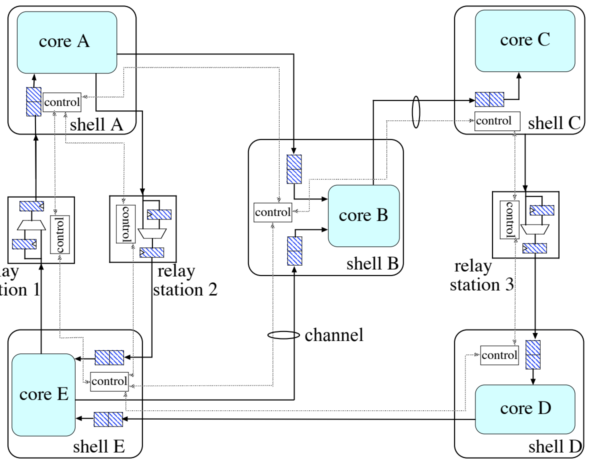


Fig. 1. Shell encapsulation, relay station insertion, and channel backpressure.

These results are made possible thanks to the separation of computation and communication, a form of orthogonalization of concerns [5], that the theory of latency-insensitive protocols formally enforces [6]. According to the LID methodology, an SoC is obtained through the assembly of *cores* (or *pearls*), each of which must be first encapsulated within an automatically-synthesized interface module called *shell* (or *wrapper*). The cores perform the actual computation in the system while the shells handle global communication and synchronization.

Figure 1 shows a latency-insensitive system with five shell-core pairs connected by point-to-point, unidirectional channels. Each core can be an arbitrarily-complex sequential module (a control logic block carrying state, a pipelined datapath with feedback loops, . . .) as long as it satisfies the requirement that it is *stallable*, i.e. it can be *clock gated*. The shell dynamically controls the operations of the core by deciding whether to stall it or *fire* it at any given clock cycle based on the value of the flow-control signals on the input/output channels. Data communicated over a channel is labeled by a bit signal indicating whether the current data is valid or void. At each clock cycle the shell fires the core if and only if each input channel presents a new valid data token (*AND-firing semantics*). Otherwise, it *stalls* the core through clock gating while storing valid data that have arrived in its input queues (for future processing) and putting void data on each output channel. Since the shell has necessarily limited storage capability, a *stop* bit signal is transmitted backward on each channel whenever a downlink shell needs to request an uplink shell to slow down the production of good data (*backpressure*).

At the implementation stage, the wires of a channel with delay longer than the target clock period can be pipelined by inserting one or more *relay stations*. A relay station is a clocked buffer with two-fold storage capacity and simple flow-control logic. By processing the void and stop bit signals, the flow-control logic of the shells and relay stations implements the latency-insensitive protocol. This is designed to accommodate arbitrary variations of delay on *inter-core* wires while guaranteeing that the functional behavior of the original synchronous system is preserved (semantics preservation) without the need of changing any part of the *intra-core* logic design [6].

LID helps to meet the required target clock frequencies through automatic wire pipelining but performance in terms of data processing throughput (number of valid data tokens processed over time) may be affected negatively by the insertion of relay stations [7], [8]. This is because each relay station that is added to the system *a posteriori* must be initialized with a void data token (a "bubble", also denoted with the symbol τ). If the relay station is inserted on a cyclic path, such as a feedback loop, the AND-firing semantics of the shells makes the bubble circulate in the loop indefinitely, thus causing the processing throughput of the overall system to drop below the ideal value (equal to one). For example, the two relay stations placed between Core A and Core E in Figure 1 induce two bubbles that circulate in the loop and stall these cores periodically, thus reducing the throughput of the entire system to 0.5. Throughput degradation can be easily computed in advance and can be reduced by optimizing the relay station insertion or the sizing of the shell queues [7], [8].

The original works on LID make a general assumption that the IP cores are *black boxes* whose internal logic structure is not known to the designers [6], [9]. These earlier works show how the knowledge of the core's I/O signals is sufficient to automatically synthesize the shell circuits. However, in assembling a complex SoC it may be the case that some cores are acquired as synthesizable modules or are developed in-house, thereby giving to the designers access to the internal details of their implementation. If indeed the core is a *white box*, then a different type of shell can be automatically synthesized around it to improve the performance of the overall latency-insensitive system. This is the topic of the present paper.

Contributions. We study how knowledge about the internal logic structure of a core can be applied to the design of its shell to improve the overall system-level performance by avoiding the unnecessary local stalling. While being fully compatible with the classic shells and relay stations, this *FIC-shell* is capable of exploiting dynamically its core's *functional independence conditions (FIC)*. Formally defined in Section II, FIC capture those scenarios when some input data are not needed for the current computation inside a core and, therefore, *even if no valid data token is present on the corresponding input channel* the core could still be fired. Such a scenario may occur for instance in a finite state machine (FSM) when it is in a certain state thereby its state transition and output functions do not depend on a given input variable. At any given clock cycle, FIC depend on the *local* logic state of the core and, potentially, on a subset of the data on some other input channels. By avoiding unnecessary stall and actually firing the core, the shell may reduce the overall number of stalls incurred in the whole system and raise its *global* processing throughput. In Section II, we present a simple motivating example of this fact, while in Section V we show its impact in a real SoC design.

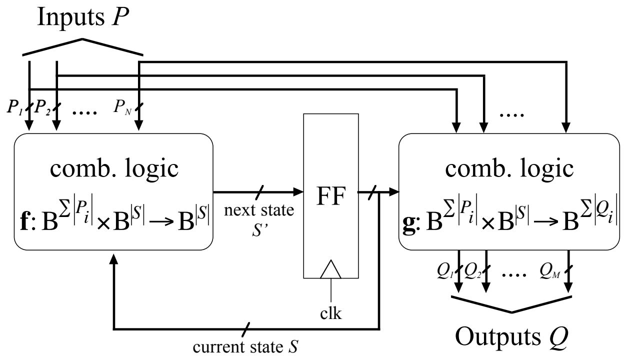

Inputs P
P_1 P_2 .... P_N
comb. logic
$\mathbf{f}: B^{\Sigma|P_i|} \times B^{|S|} \rightarrow B^{|S|}$
next state S'
FF
clk
comb. logic
$\mathbf{g}: B^{\Sigma|P_i|} \times B^{|S|} \rightarrow B^{\Sigma|Q_i|}$
....
Q_1 Q_2 .... Q_M
Outputs Q
current state S

Fig. 2. Modeling a core module as a finite state machine (FSM).

In Section III we present a novel circuit design of a generic FIC-shell that can dynamically exploit FIC when the core is given as a *white box*. Like for the original simpler shell in LID, this design can be used as a parameterized template to automatically synthesize a specific instance of the FIC-shell for any given stallable core.

In Section IV we provide a *fully automatic* procedure for the logic synthesis of the main logic block of a FIC-shell instance based on the particular characteristics of its corresponding core. Our method requires no input from designers and relies on efficient logic synthesis algorithms.

In Section V we analyze in detail the applicability and effectiveness of the performance optimization based on FIC in the LID methodology. This includes a report on the semi-custom design of a real SoC using LID. Our results confirm that the system performance of a latency-insensitive system can benefit considerably from this idea with minor area (and no delay) overhead.

Finally, in Section VI we present an extensive discussion of related work.

II. Functional Independence Conditions

Without loss of generality a core can be viewed as synchronous logic network [10] and can be modeled as a finite state machine (FSM). We revisit the classic FSM model in the context of LID to highlight the role played by the core's I/O channels (Figure 2):

- The inputs of the FSM is a set of Boolean variables, partitioned into N groups: $P = P_1 \bigcup \ldots \bigcup P_N$, where P_i is a set of w_i Boolean variables $\{p^i_1, \ldots, p^i_{w_i}\}$, representing the data portion of an input channel i of parallelism w_i.
- The outputs of the FSM is a set of Boolean variables partitioned into M groups: $Q = Q_1 \bigcup \ldots \bigcup Q_M$, where $Q_j = \{q^j_1, \ldots, q^j_{w_j}\}$ represents the data of an output channel j of parallelism w_j.

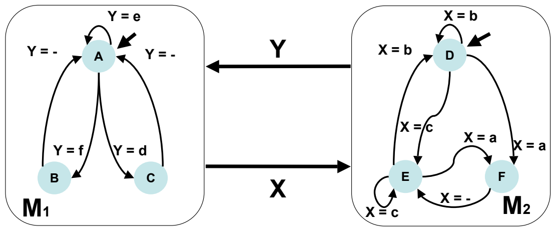


Fig. 3. A synchronous system made of two communicating FSMs.

- Let $S = \{s_1, \ldots, s_n\}$ and $S' = \{s'_1, \ldots, s'_n\}$ be the sets of Boolean variables representing the FSM present-state and next-state, respectively. At each clock transition the next-state's values becomes the present-state's values.
- Let $\mathsf{B} = \{0, 1\}$. The *state transition functions* are an array of Boolean functions mapping the input and present-state variables to the next-state variables $f_i : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}$. We also simply write $\mathbf{f} : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}^{|S|}$ Likewise, the *output functions* are an array of Boolean functions mapping the input and present-state variables to the output variables $g_j : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}$, or simply $\mathbf{g} : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}^{|Q_1|+\cdots+|Q_M|}$.

We now give a definition of FIC based on the FSM model:

Definition 1 *Let* $\mathrm{T} \equiv \{\widetilde{\mathrm{P_1}}, \ldots, \widetilde{\mathrm{P_k}}, \ldots, \widetilde{\mathrm{P_N}}; \widetilde{\mathrm{S}}\}$ *be a tuple of values for the input and present state of a FSM; the state transition functions and output functions are* independent *from value* $\widetilde{\mathrm{P_k}}$ *of channel* P_k *when for* any other *tuple of values* $\mathrm{T}' \equiv \{\widetilde{\mathrm{P_1}}, \ldots, \widetilde{\mathrm{P'_k}}, \ldots, \widetilde{\mathrm{P_N}}; \widetilde{\mathrm{S}}\}$ *that only differs for the value of input channel* P_k*, we have*

$$\mathbf{f}(\mathrm{T}) = \mathbf{f}(\mathrm{T}') \ \ \textit{and} \ \ \mathbf{g}(\mathrm{T}) = \mathbf{g}(\mathrm{T}') \qquad (1)$$

Whether $\mathbf{f}$ and $\mathbf{g}$ are independent from the value of an input channel is contingent on the values of the other input channels and the present state. Given a tuple $\mathrm{T} \equiv \{\widetilde{\mathrm{P_1}}, \ldots, \widetilde{\mathrm{P_k}}, \ldots, \widetilde{\mathrm{P_N}}; \widetilde{\mathrm{S}}\}$, of input and present-state values, if $\mathbf{f}$ and $\mathbf{g}$ are independent from the value of P_k, we call [1]

$$FIC_{P_k}(\mathrm{T}) \equiv \{\widetilde{\mathrm{P_1}}, \ldots, \widetilde{\mathrm{P_{k-1}}}, \widetilde{\mathrm{P_{k+1}}}, \ldots, \widetilde{\mathrm{P_N}}; \widetilde{\mathrm{S}}\} \qquad (2)$$

a *functional independence condition* of input channel P_k.

Generally, there may be more than one tuple of input and present-state values under which the core's computation is independent from the value of input channel P_k.

Definition 2 *Let* $\mathcal{T} \subseteq \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^n$ *be the set of all possible tuples of input and present-state values. The complete set of functional Independence condition (FIC) of channel* P_k *is*

$$FIC_{P_k} \equiv \bigcup_{\mathrm{T} \in \mathcal{T}} FIC_{P_k}(\mathrm{T}) \qquad (3)$$

Since the number of the distinct input and present-state values is finite, the set FIC_{P_k} is also finite.

[1]We use the term FIC instead of *don't care* because the latter should be reserved for those input minterms of a Boolean function for which the function's output value is not specified or not needed [10], [11].

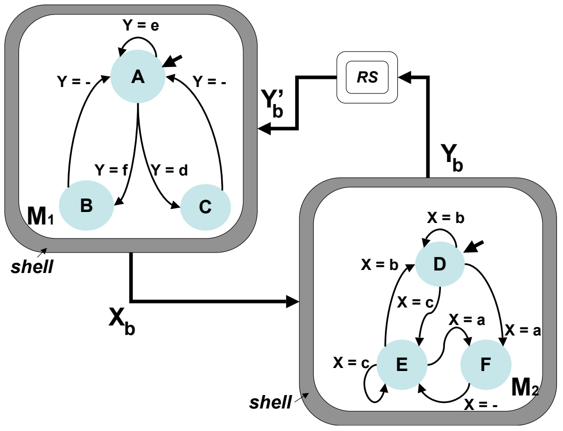


Fig. 4. A latency-insensitive system derived from the system of Fig. 3

Next, we present a simple example to show how FIC can be used to optimize the performance of a latency-insensitive system. The realization of such optimization hinges on the identification of FIC. In Section IV we provide a procedure to find FIC for each input channel. The set of FIC returned by our procedure is implicitly represented as a Boolean predicate that can be efficiently implemented as a hardware logic block (the FIC-detect block), which in turn becomes part of the FIC-shell.

A. Motivating Example

Consider the synchronous system of Figure 3 having two interconnected Moore FSMs M_1 and M_2. Each FSM has a single input variable that is set equal to the output variable of the other FSM: X is the output of M_1 and the input of M_2, while Y is the output of M_2 and the input of M_1. In the FSM state transition diagrams each edge is labeled with the value of the input variable that activates the corresponding transition. Both FSMs have three states: the set of states of M_1 is $\{A, B, C\}$ and the set of states of M_2 is $\{D, E, F\}$. Since we have single-output Moore FSMs, we simply assume that in each state S the value of the output variable is equal to the corresponding lowercase letter s: in other words, FSM M_1 outputs $X = a$ while being in state A, $X = b$ while in state B, and $X = c$ while in state C. Similarly, FSM M_2 outputs $Y = d$ while being in state D, $Y = e$ while in state E, and $Y = f$ while in state F. As denoted by the arrow, the initial states are respectively A for M_1 and D for M_2. There are three sets of traces in Figure 5: the first set captures the behavior of the strictly synchronous system of Figure 3. Notice that the system cycles through five compound state transitions: for M_1 we have $(A \rightarrow C \rightarrow A \rightarrow A \rightarrow B) \rightarrow (A \rightarrow C \ldots)$, while for M_2 we have $(D \rightarrow F \rightarrow E \rightarrow F \rightarrow E) \rightarrow (D \rightarrow F \ldots)$.

The second set of traces in Figure 5 describes the behavior of the system of Figure 4: this latency-insensitive system is obtained from the system of Figure 3 by encapsulating each FSM with a distinct shell and inserting a relay station on the channel from M_2 to M_1. Since the relay station is initialized with a void token (denoted as τ), this is what variable Y'_b presents at the first cycle t_0. Due to the AND-firing semantics of LID, this value continues to iterate in the feedback loop

		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	…
Strict System	X:	a	c	a	a	b	a	c	a	a	b	a	c	a	a	b	a	…
	Y:	d	f	e	f	e	d	f	e	f	e	d	f	e	f	e	d	…
LI System (black boxes)	X_b:	a	$\boxed{\tau}$	c	a	$\boxed{\tau}$	a	b	$\boxed{\tau}$	a	c	$\boxed{\tau}$	a	a	$\boxed{\tau}$	b	a	…
	Y'_b:	$\boxed{\tau}$	d	f	$\boxed{\tau}$	e	f	$\boxed{\tau}$	e	d	$\boxed{\tau}$	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	…
	Y_b:	d	f	$\boxed{\tau}$	e	f	$\boxed{\tau}$	e	d	$\boxed{\tau}$	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	…
	stalling:	M_1	M_2	–	M_1	M_2	–	M_1	M_2	–	M_1	M_2	–	M_1	M_2	–	M_1	…
LI System (white boxes) after FIC-based optimization	X_b:	a	$\boxed{\tau}$	c	a	a	$\boxed{\tau}$	b	a	$\boxed{\tau}$	c	a	a	$\boxed{\tau}$	b	a	$\boxed{\tau}$	…
	Y'_b:	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	…
	Y_b:	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	…
	stalling:	M_1	–	(M_2)	–	M_1	M_2	–	M_1	–	(M_2)	–	M_1	M_2	–	M_1	–	…

Fig. 5. Set of traces for the behaviors of the three systems in the motivating example.

forcing each shell to periodically stall its core FSM: M_1 stalls at t_{3n} while M_2 stalls at t_{3n+1} with $n \geq 0$. Pairwise comparison of the X, Y traces with the X_b, Y_b traces shows that they are *latency-equivalent* as expected [6]: i.e., they are the same if one ignores the τ symbols. But, the data processing throughput of the system is reduced from 1 to $\frac{2}{3} = 0.66$.

Part of the lost throughput, however, can be recovered if one takes advantage of FIC by analyzing the internal structure of the FSM (an assumption not made in [6] where cores are treated as *black boxes*). For instance, when M_2 is in state F, its computation is independent from the value of input channel X_b. Thus, the present-state value F is a FIC of X_b under all possible input patterns: $FIC_{X_b} \equiv \{*; S_{M_2} = F\}$. This FIC can be used to design a shell that: (a) avoids to stall M_2 whenever it is in state F *and* there is a τ on channel X_b (*stall avoidance*); (b) remembers that after each stall avoidance it must eventually stall M_2 when the "previously-unneeded" data on channel X_b arrives, only to be discarded (*delayed stall*). This is what happens first at cycles (t_1, t_2) and then again at cycles (t_8, t_9) in the third set of traces of Figure 5 where the stalled FSM is reported in the last row (and delayed stalls are marked with parenthesis). The key point is that, even for this simple system, delaying a local stall by a single clock cycle allows us to raise the global throughput by 9% to $\frac{5}{7} = 0.72$.

III. Shell Design

We present the design of a shell interface module that can exploit functional independence conditions (*FIC-shell*). This is a variation of the shell design presented in [9], [12], which we review first.

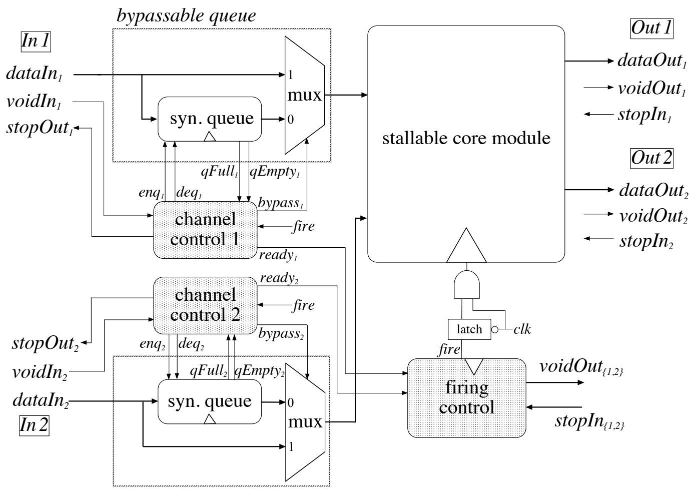
A. Classic Shell With Backpressure

A *classic* shell aligns the incoming data tokens, which may arrive with arbitrary latencies, so that the input and output traces of an encapsulated core module is *latency-equivalent* to the original core module. Conceptually a shell has two different kinds of logic controllers (though in implementation they can be combined): a *firing control block* decides when a core module should be stalled by gating the core's clock and a *channel control block* handles incoming data tokens, interface signals, and input queue operations for each channel. Figure 6(a) reports a block diagram of a two-input-two-output classic shell. A shell receives data from input channels and broadcasts outputs of the core to output channels at every clock cycle. A channel carries data and two special 1-bit signals: *void* and *stop*. The void signal is used by the sender shell to inform the sender's downlink receivers whether the accompanying data is valid. The stop signal is a flow-control signal and is used by a receiver to inform the receiver's uplink sender to stop sending more data (backpressure).

The shell control logic is presented in Figure 6(b) [9], [12]. At each clock cycle the shell decides whether the computation of a core module can proceed: the computation is allowed for the next clock cycle ("firing") if and only if the *fire* signal is high (Eq. 4). The signal *fire* is set high if all of the input channels are *ready*, and no downlink receiver sends in backpressure. Otherwise the shell stalls the core by setting *fire* low to gate the core's clock. In a classic shell an input channel i is ready if it presents a valid data token, either from the channel ($voidIn_i = 0$) or from the queue ($qEmpty_i = 0$), as stated in Eq. 7. The signal $backpressure_j$ is set high if a downlink receiver of an output channel j is unable to save a valid data generated by the sender in its own queue; it is indicated by Eq. 5 with $stopIn_j = 1$ and $voidOut_j = 0$. The output tokens generated by a stalled module are marked as void if there is no backpressure from the receivers (the second clause in Eq. 6). In the case of backpressure from a downlink receiver of an output channel j, the data to the receiver is marked as valid until the receiver has storage space to save it (the first clause in Eq. 6) because a void token is never saved by the shell of the receiver. Eq. 8 to 10 are the rules for steering the input data. For an input channel i, a valid but not consumed (due to stalling) data is stored in its queue for later use, indicated by $enq_i = 1$ as in Eq. 8. If the core is fired and the queue is not empty, the data at the head of the queue is used by the core and thus it is dequeued ($deq_i = 1$, Eq. 9). The $bypass_i$ signal directs the proper data to the core, either from the channel or from the output of the queue (Eq. 10). Finally, when the queue is full ($qFull_i = 1$), the $stopOut_i$ is set high, thus activating the backpressure to request the uplink sender of the input channel to stall (Eq. 11).

B. Design of a FIC-Shell

Figure 7(a) reports a block diagram of the newly proposed FIC-shell design. While the firing control block of the classic shell is reused, the channel control logic is modified to support the new *stall avoidance* and *delayed stall* operations discussed in the example in Section II. First, the FIC-shell differs from the classic shell by the conditions deciding a channel's *readiness*. Normally a FIC-shell operates like a classic shell, but it "becomes more aggressive" when FIC can be exploited, i.e. whenever one or more input channels present invalid data which are not necessary to the core's computation. In

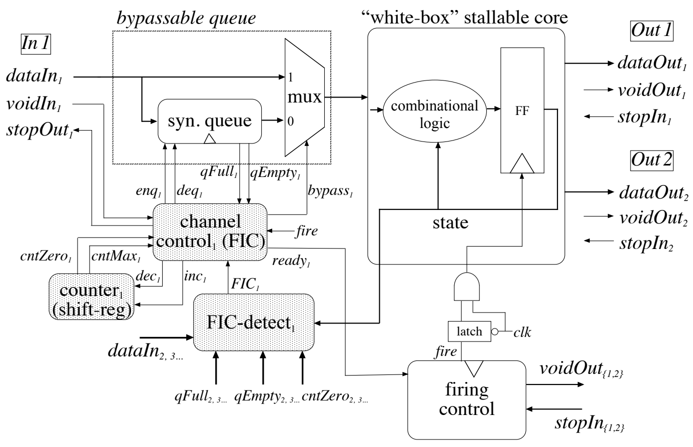


(a)

$$\begin{aligned}
fire &= \bigwedge_{i\in\mathcal{I}}(ready_i)\cdot\overline{\bigvee_{j\in\mathcal{O}} backpressure_j} && (4)\\
\forall j\in\mathcal{O}\ backpressure_j &= stopIn_j\cdot\overline{voidOut_j} && (5)\\
\forall j\in\mathcal{O}\ {voidOut_j}^{+} &= \begin{cases} 0 & \text{if } backpressure_j \text{ is true}\\ \overline{fire} & \text{otherwise}\end{cases} && (6)\\
\forall i\in\mathcal{I}\ ready_i &= \overline{voidIn_i}+\overline{qEmpty_i} && (7)\\
\forall i\in\mathcal{I}\ enq_i &= \overline{voidIn_i}\cdot(\overline{fire}+\overline{qEmpty_i})\cdot\overline{qFull_i} && (8)\\
\forall i\in\mathcal{I}\ deq_i &= \overline{qEmpty_i}\cdot fire && (9)\\
\forall i\in\mathcal{I}\ bypass_i &= qEmpty_i && (10)\\
\forall i\in\mathcal{I}\ stopOut_i &= qFull_i && (11)
\end{aligned}$$

(b)

Fig. 6. (a) A block diagram of a two-input-two-output shell and a stallable core module. (b) The channel and firing control logic of the shell.



(a)

$$\begin{aligned}
fire &= \bigwedge_{i\in\mathcal{I}}(ready_i)\cdot\overline{\bigvee_{j\in\mathcal{O}} backpressure_j} && (12)\\
\forall j\in\mathcal{O}\ backpressure_j &= stopIn_j\cdot\overline{voidOut_j} && (13)\\
\forall j\in\mathcal{O}\ {voidOut_j}^{+} &= \begin{cases} 0 & \text{if } backpressure_j \text{ is true}\\ \overline{fire} & \text{otherwise}\end{cases} && (14)\\
\forall i\in\mathcal{I}\ ready_i &= \overline{qEmpty_i}+\overline{voidIn_i}\cdot cntZero_i +\\
&\quad FIC_i\cdot(\overline{cntMax_i}+\overline{voidIn_i}) && (15)\\
\forall i\in\mathcal{I}\ enq_i &= cntZero_i\cdot\overline{voidIn_i}\cdot(\overline{fire}+\overline{qEmpty_i})\cdot\overline{qFull_i} && (16)\\
\forall i\in\mathcal{I}\ deq_i &= \overline{qEmpty_i}\cdot fire && (17)\\
\forall i\in\mathcal{I}\ inc_i &= fire\cdot voidIn_i\cdot qEmpty_i\cdot\overline{cntMax_i} && (18)\\
\forall i\in\mathcal{I}\ dec_i &= \overline{voidIn_i}\cdot\overline{fire}\cdot\overline{cntZero_i} && (19)\\
\forall i\in\mathcal{I}\ bypass_i &= qEmpty_i && (20)\\
\forall i\in\mathcal{I}\ stopOut_i &= qFull_i && (21)
\end{aligned}$$

(b)

Fig. 7. (a) Block diagram of a FIC-shell. (b) FIC-shell channel and firing control logic. For simplification we only show one input channel with $i = 1$.

this case, these channels are declared ready and the FIC-shell fires the core module. However, this operation makes the core run one more clock cycle *ahead* of the next valid data for such channels. So, when this data arrives it must be discarded. Therefore, for each input channel a FIC-shell maintains a counter that records how many cycles the core module currently runs ahead with respect to the next valid data on the channel.

For an input channel i, whether a FIC for the channel is satisfied at a given clock cycle is dynamically established by the *FIC-detect$_i$* block: this is a combinational logic block that monitors the present state of the core and the values of other input channels. Each channel has its own single-output FIC-detect block [2]. When the *FIC-detect$_i$* sets FIC$_i$ to high, the current data of the channel is not needed for the core's computation. In Section IV we present a procedure for the logic synthesis of this block.

Figure 7(b) lists the channel and firing control logic of the FIC-shell. As indicated earlier, the firing control logic (Eq. 12 to Eq. 14) is the same as the one of the classic shell. On the other hand, the channel control logic is different because it takes advantage of FIC for potential stall avoidance and updates counters of input channels to induce delayed stalls. The control logic of an input channel i follows simple rules implemented as Eq. 18 and Eq. 19 to maintain the count of the input channel: whenever a core is fired but the input channel has no valid data (i.e. it receives void data $voidIn_i = 1$ and the queue is empty $qEmpty_i = 1$), the count of the channel is incremented by 1 (Eq. 18). A non-zero count indicates the next valid data is outdated, and the valid data should be discarded on arrival (potentially causing a delayed stall). When a delayed stall is caused by dropping a valid but outdated data in this case, the count is decreased by 1 (Eq. 19). The channel control also decides the readiness of an input channel following Eq. 15. An input channel i is ready if any of the following conditions holds:

1) The queue provides a valid data ($qEmpty_i = 0$).
2) The channel provides a valid ($voidIn_i = 0$) and fresh data (marked by the zero count, i.e. $cntZero_i = 1$).

[2]In practice, all the FIC-detect blocks can be combined into a single component to increase logic optimization opportunities.

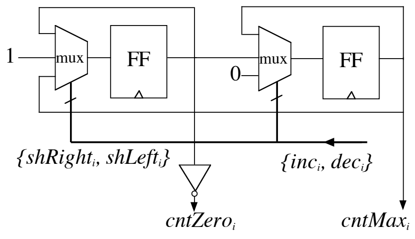


Fig. 8. A shift register used as the counter in the FIC-shell.

3) The core's computation does not depend on the data value (indicated by $FIC_i = 1$), and either the counter has not reached its maximum value ($cntMax_i = 0$) or the data is valid ($voidIn_i = 0$). When the count reaches its maximum value, the channel control can no longer declare its channel as ready even if this channel is receiving a void data and the FIC_i is true, because exploiting the FIC for stall avoidance in such cases would cause the overflow of the counter. However, there is an exception to this rule: When the counter reaches its maximum value, an input channel is still declared as ready if a *valid* ($voidIn_i = 0$) data is received and the core's computation is independent from the data ($FIC_i = 1$). This is because the count will not be increased in such cases: this valid data is dropped regardless of whether the core will be fired, so the count will either be the same (if the core is fired) or will be decreased (not fired). So regardless of the maximum value of the count, the FIC-shell will always be able to synchronize the incoming data properly.

In practice, instead of using an up-down counter, a shift register is sufficient because the actual count is not needed. Figure 8 shows an implementation of a 1-bit wide shift register which is used as a counter in our FIC-shell. Increasing the count by one shifts a "1" into the register ($shRight_i$, which is inc_i); decreasing the count by one shifts a "1" out ($shLeft_i$, which is dec_i). Evaluating $cntZero_i$ and $cntMax_i$ is straightforward: the inverse of the leftmost bit indicates whether the count is zero ($cntZero_i$), and the rightmost bit flags whether the register reaches its maximum capacity ($cntMax_i$).

We use purely synchronous queues to save unused but valid data tokens. The enqueuing and dequeuing operations of a queue only take effect at the rising (or falling) clock edges, i.e. the enqueued data token is latched by the queue's storage element at the next rising/falling clock edge. Similarly, a queue updates its output, including the data and queue status signals ($qFull_i$ and $qEmpty_i$) at every rising/falling clock edge. The synchronous queue implementation has no combinational path between its inputs and outputs.

Remark. A FIC-shell still follows the latency-insensitive protocol when it communicates with relay stations or other shells. It only relaxes the conditions of firing a core module. So, *FIC-shells and classic shells can co-exist in a system.* Therefore a designer can use FIC-shells only when it is beneficial to the system's performance. Since the system throughput is set by the critical feedback loops [7], [8], FIC-shells can be used only for those core modules that are part of such loops, while classic shells are sufficient elsewhere.

C. The FIC-Queue: Reducing the Stalls due to Backpressure

FIC can also be used to reduce the stalls due to backpressure. This can be achieved by "virtually" increasing the queue sizes without allocating real storage elements for data. To do so, we designed a new queue, called *FIC-queue* [3]. The FIC-queue maintains the same operating semantics as a normal synchronous queue. Internally, for any data which is not needed for the core's computation the queue only remembers the data's existence but not the value. In such cases, compared to normal synchronous queues with the same amount of data storage elements, the new FIC-queue appears to be larger. Thus the FIC-queue can potentially reduce the number of stalls caused by backpressure.

Figure 9 reports the implementation of the FIC-queue design taking advantage of FIC. The FIC-queue replaces the original queue in Figure 7(a), and provides the same semantics as discussed earlier. Figure 9(a) shows the block diagram of the queue, its internal signals, and its external interface with the remaining logic of the shell. Compared to the original queue, the FIC-queue of an input channel i reads one more input signal FIC_i, which indicates whether the core's computation is independent from the oldest not used data on the input channel [4]. The control examines the status of the internal queue and the FIC_i signal to decide whether the oldest data should be saved. The control logic is implemented as a two-state FSM. Figure 9(b) shows the state transition diagram of the FSM and the values of outputs at the two states. Initially the control is in the state NOT_FI. The FI state indicates that a data not needed for the core's computation exists but its value is discarded. The control discards a data not critical to the core's computation and enters the FI state in either of the following two scenarios:

1) If the queue is empty and the core's computation does not depend on the input data ($FIC_i = 1$) and the channel control logic enqueues the data ($\overline{deq_i} \cdot enq_i$), then the control discards the data and enters the FI state.
2) If the core's computation is independent from the head of the internal queue ($FIC_i \cdot \overline{deq_i} \cdot \overline{qEmpty_i}$), then it is popped out and the FSM enters the FI state.

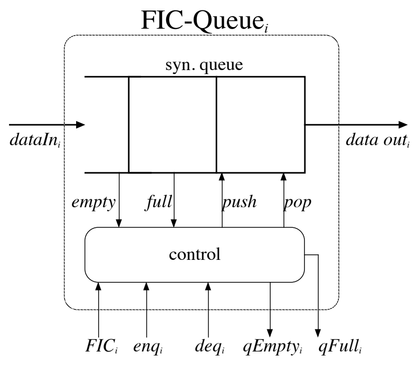
At the FI state, the FSM reports externally that the queue is not empty, regardless of the status of the internal queue. Note that $qFull_i$ and $qEmpty_i$ are both sequential signal as they depend only on the internal *full* and *empty* signals, which are updated at rising/falling clock edges, as discussed earlier.

IV. Logic Synthesis of FIC-Detect Block

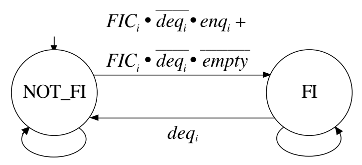
We present a procedure to automatically identify the set of functional independence conditions (FIC) as defined in Section II. The FIC are returned as logic predicates of present state and current input variables; they can be implemented as simple combinational logic. We then use the FIC to synthesize

[3]As discussed in Section VI, the FIC-queue generalizes a technique recently proposed in [13].

[4]This is either the incoming data from the channel if the internal queue is empty, or the head of the internal queue if it is not empty.



(a)



$FIC_i \bullet \overline{deq_i} \bullet enq_i +$
$FIC_i \bullet \overline{deq_i} \bullet \overline{empty}$
NOT_FI
FI
deq_i

output signal	state NOT_FI	state FI
$push$	$\begin{cases} 0 & \text{if } FIC_i \cdot empty \cdot \overline{deq_i} \cdot enq_i \\ enq_i & \text{otherwise} \end{cases}$	enq_i
pop	$\begin{cases} 1 & \text{if } FIC_i \cdot \overline{deq_i} \cdot \overline{empty} \\ deq_i & \text{otherwise} \end{cases}$	0
$qFull_i$	$full$	$full$
$qEmpty_i$	$empty$	0

(b)

Fig. 9. FIC-queue for a given input cahnnel i: (a) block diagram; (b) FSM and its output signals of the control in (a).

the FIC-detect block for the channel. Our procedure builds on the unobservability conditions of a Boolean function to derive FIC [5]. First, we recall some background concepts.

A. Background Definitions

For a Boolean function f, a variable x is *unobservable* if f is not sensitive to the changes of x [10]. A variable's unobservability may only hold under certain conditions that are expressed by the complement of the *Boolean difference*, which computes under what conditions f is sensitive to x. The Boolean difference is simply the result of *XOR* ($\oplus$) of f's co-factor with respect to x and $\overline{x}$. Thus, the conditions under which function f is insensitive to variable x is:

$$\overline{\frac{\partial f}{\partial x}} = f|_{x=1} \overline{\oplus} f|_{x=0}$$

where $\overline{\oplus}$ is the complement of XOR.

The *consensus* of Boolean function f with respect to variable x is the part of f that is independent of x:

$$C_x(f) = f|_{x=1} \cdot f|_{x=0} \tag{22}$$

Consensus can be extended to a set of variables by iteratively applying Eq. 22 to each variable [10].

B. Logic Synthesis of the FIC-Detect Block

We now introduce our procedure, which consists of four steps. Our presentation is based on the FSM model of a core and the definition of FIC as given in Section II.

Step 1. To derive the FIC for an input channel P_i, we first restrict the computation to a single Boolean input variable $p^i_k \in P_i$ with respect to a scalar state transition function $f_{s'_t}$ ($s'_t \in S'$ is a single next state variable). We have:

$$\overline{\frac{\partial f_{s'_t}}{\partial p^i_k}} = f_{s'_t}|_{p^i_k=1} \overline{\oplus} f_{s'_t}|_{p^i_k=0} \tag{23}$$

Similarly for the unobservability of p^i_k w.r.t. an output function g^j_ℓ for an output variable $q^j_\ell \in Q_j$ we have:

$$\overline{\frac{\partial g^j_\ell}{\partial p^i_k}} = g^j_\ell|_{p^i_k=1} \overline{\oplus} g^j_\ell|_{p^i_k=0} \tag{24}$$

Computing unobservability conditions using Boolean difference directly on a large multi-level Boolean network may not be practical, unless the network's global logic functions $\mathbf{f}$ and $\mathbf{g}$ are given, or can be efficiently derived. An effective solution, which has been shown successful on large designs, is to iteratively applying Boolean difference locally. In addition, methods of approximating unobservability conditions to reduce the size of their representations can also be applied [10] [6]. For simplicity, in the sequel we denote the computation of unobservability conditions with the Boolean difference operator.

Step 2. Since FIC involve all state and output variables we perform the conjunction of all the unobservability conditions by Eq. (23) and Eq. (24):

$$\widetilde{FIC}_{p^i_k}(\mathbf{f},\mathbf{g}) = (\bigwedge_{s'_t \in S'} \overline{\frac{\partial f_{s'_t}}{\partial p^i_k}}) \cdot (\bigwedge_{Q_i \in Q} \bigwedge_{g^j_\ell \in Q_i} \overline{\frac{\partial g^j_\ell}{\partial p^i_k}}) \tag{25}$$

Step 3. A channel P_i has generally many input variables. Hence, we take the conjunction across:

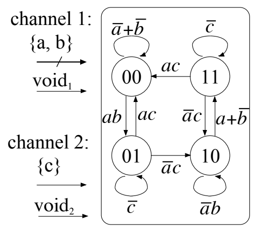
$$\begin{aligned} \widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g}) &= C_{P_i}(\bigwedge_{p^i_k \in P_i} \widetilde{FIC}_{p^i_k}(\mathbf{f},\mathbf{g})) \\ &= C_{p_{i_1}}(C_{p_{i_2}}(\cdots C_{p^i_k}(\bigwedge_{p^i_k \in P_i} \widetilde{FIC}_{p^i_k}(\mathbf{f},\mathbf{g}))\cdots) \end{aligned} \tag{26}$$

Note that the consensus function is used to eliminate any cube that contains input variables from channel P_i. These cubes can arise after taking the conjunctions across the unobservability conditions of single variables.

Step 4. In LID not every input channel presents a valid data at each clock cycle. So we require all the input variables which appear in Eq. (26) to come from input channels presenting valid data. Recall that a good data can come either from the channel (i.e. its $voidIn$ is 0) or from the channel's queue (i.e. the queue is not empty $qEmpty = 0$). Further, if the data is

[5]Computing unobservability conditions is used to derive *observability don't cares*. It is the basis of our procedure, but not the focus of this paper. We refer the interested reader to [10].

[6]If some approximations are used, our procedure returns a subset of FIC defined in Section II.



state transition functions:

$$f_0 = s_1\overline{a}c + s_0\overline{c} + s_0\overline{s_1}$$
$$f_1 = \overline{s_1}ab + s_0\overline{s_1}\overline{b} + s_1\overline{c}$$

(a)

$$\begin{aligned}
\overline{\frac{\partial f_0}{\partial a}} &= \overline{s_1} + \overline{c}\\
\overline{\frac{\partial f_1}{\partial a}} &= s_1 + \overline{b}\\
\overline{\frac{\partial f_0}{\partial b}} &= 1\\
\overline{\frac{\partial f_1}{\partial b}} &= s_1 + s_0a + \overline{s_0a}\\
\overline{\frac{\partial f_0}{\partial c}} &= \overline{s_1} + s_0\overline{a} + \overline{s_0}a\\
\overline{\frac{\partial f_1}{\partial c}} &= \overline{s_1}
\end{aligned}$$

(b)

$$\boxed{\begin{aligned}
FIC^{\mathtt{IS}}_{P_1}(\mathbf{f}) &= s_1\overline{c}\,(\overline{voidIn_2}\cdot cntZero_2 + \overline{qEmpty_2})\\
FIC^{\mathtt{IS}}_{P_2}(\mathbf{f}) &= \overline{s_1}
\end{aligned}}$$

(c)

$$\boxed{\begin{aligned}
FIC^{\mathtt{S}}_{P_1}(\mathbf{f}) &= \emptyset\\
FIC^{\mathtt{S}}_{P_2}(\mathbf{f}) &= \overline{s_1}
\end{aligned}}$$

(d)

Fig. 10. (a) A core module with 2 input channels (3 input variables in total) modeled by a 4-state Moore FSM. (b) The unobservability conditions of each input variable with respect to the 2-state transition functions. (c) The final FIC depending both on inputs and states. (d) The final FIC depending only on states.

from the channel, it cannot be outdated (the count is zero, i.e. $cntZero = 1$). So the final FIC can be obtained as follows:

$$FIC^{\mathtt{IS}}_{P_i}(\mathbf{f},\mathbf{g}) \equiv \text{Replace each literal } p \text{ in } \widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g}) \text{ with } p\cdot(\overline{voidIn_k}\cdot cntZero_k + \overline{qEmpty_k}), \text{ and } \overline{p} \text{ with } \overline{p}\cdot(\overline{voidIn_k}\cdot cntZero_k + \overline{qEmpty_k}) \tag{27}$$

where $voidIn_k$, $qEmpty_k$, and $cntZero_k$ are signals from the input channel k containing the variable p.

The domain of the single-output Boolean function $FIC^{\mathtt{IS}}_{P_i}(\mathbf{f},\mathbf{g})$ that is obtained at the end of Step 4 is the set of state variables, input variables, $voidIn$ and $qEmpty$ variables minus the set of input, $voidIn_i$ and $qEmpty_i$ variables of the channel P_i. A combinational logic network can be synthesized to implement this function within the channel FIC-detect block: at each clock cycle, if $(FIC^{\mathtt{IS}}_{P_i}(\mathbf{f},\mathbf{g}) = 1)$ then the current data value of the channel P_i is not needed to compute the state transition and the output function of the core.

The firing of a core module is controlled by the *fire* signal, which must be stable by the end of each clock cycle. Therefore, the dependency of FIC on input-channel variables may induce extra timing constraints as it may lead to long combinational paths from an uplink sender of data to the *fire* signal across the communication channel. Hence, we may want to restrict ourselves to FIC depending only on state variables. This requires a different (alternative) final step in our procedure.

Step 4'. To restrict FIC dependency to state variables only, we apply the consensus function to Eq. (26) over all input variables iteratively:

$$\begin{aligned}
FIC^{\mathtt{S}}_{P_i}(\mathbf{f},\mathbf{g}) &= C_P(\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g}))\\
&= C_{p_{1_1}}(C_{p_{1_2}}(\cdots C_{p^i_k}(\cdots(\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})\cdots))) \qquad (28)
\end{aligned}$$

If the core module has no combinational path from its inputs to outputs (thus it can be viewed as a Moore FSM), Eq. (24) will return 1 because an output variable does not depend on any input. The same steps can still be applied to make $FIC^{\mathtt{S}}_{P_i}(\mathbf{f},\mathbf{g})$ equal to $FIC^{\mathtt{S}}_{P_i}(\mathbf{f})$.

Example. We apply the procedures discussed above to a simple core module whose behavior is modeled by a Moore FSM. The core, its FSM model, and the state transition functions are reported in Figure 10(a). It has two input channels consisting of three variables in total ($\{a,b\}$ and c), and four states encoded as $(s_0s_1) \in \{00, 01, 10, 11\}$.

We applied our four-step procedures to derive the FIC for each input channel. Since the core is a Moore FSM, only Eq. (23) must be applied in Step 1. The FIC of all three input variables with respect to each state transition function are shown in Figure 10(b). Finally, Eq. (25) and Eq. (26) provide the FIC for each of the two channels: $FIC^{\mathtt{IS}}_{P_1}(\mathbf{f}) = s_1\overline{c}\,(\overline{voidIn_2}\cdot cntZero_2 + \overline{qEmpty_2})$ and $FIC^{\mathtt{IS}}_{P_2}(\mathbf{f}) = \overline{s_1}$.

If we prefer to restrict ourselves to FIC depending only on the state variables, then we apply Step 4' instead of Step 4. In this case, the FIC for Channel 2 becomes $FIC^{\mathtt{S}}_{P_2}(\mathbf{f}) = \overline{s_1}$, while the input data coming at Channel 1 are always needed: $FIC^{\mathtt{S}}_{P_1}(\mathbf{f}) = \emptyset$. Overall, less opportunities for avoiding stalling can be exploited, but this might be necessary to meet timing constraints on the shell logic.

V. Experimental Results

We present various experiments designed to evaluate the applicability, efficiency, and overhead of the proposed optimization technique. We implemented the FIC-computation procedure discussed in Section IV within the logic synthesis tool ABC [14]. We test it with a suite of sequential circuits including the ISCAS-89 benchmarks, and with a real-world SoC, an ultra-wideband baseband transmitter [15], [16]. Both experiments demonstrate that FIC-based optimization has broad applicability, is efficient, and imposes little overhead.

A. Applicability of FIC optimizations

In the first set of experiments, we evaluate the applicability and practicality of FIC optimization by applying it to ISCAS-89 benchmarks and other sequential circuits. For each benchmark, the FIC are derived assuming that each single input is a LID channel (this overly-simplified assumption will be later discarded when we apply FIC optimization to the SoC). We distinguish a FIC that depends only on core's state variables (SD-FIC) from one that depends also on input variables (ISD-FIC). Figure 11 reports three distributions showing the occurrence frequencies of FIC in reachable states for benchmark circuit *s1488*. Figure 11(a) lists the ratio of reachable states in which a particular input has FIC. Figure 11(b) lists the number of inputs which have FIC in each of the 48 reachable states. Figure 11(c) shows the ratio of states where at least some inputs have SD-FIC. Note that the analysis only considers *satisfied* SD-FIC at each reachable state for a given input. In benchmark circuit *s1488*, SD-FIC are very frequent: all but two inputs have satisfied SD-FIC in most states. Further, in most reachable states there is a significant number of inputs which have FIC. Note that SD-FIC dominate, and by considering also ISD-FIC only a little more functional independence conditions can be exploited.

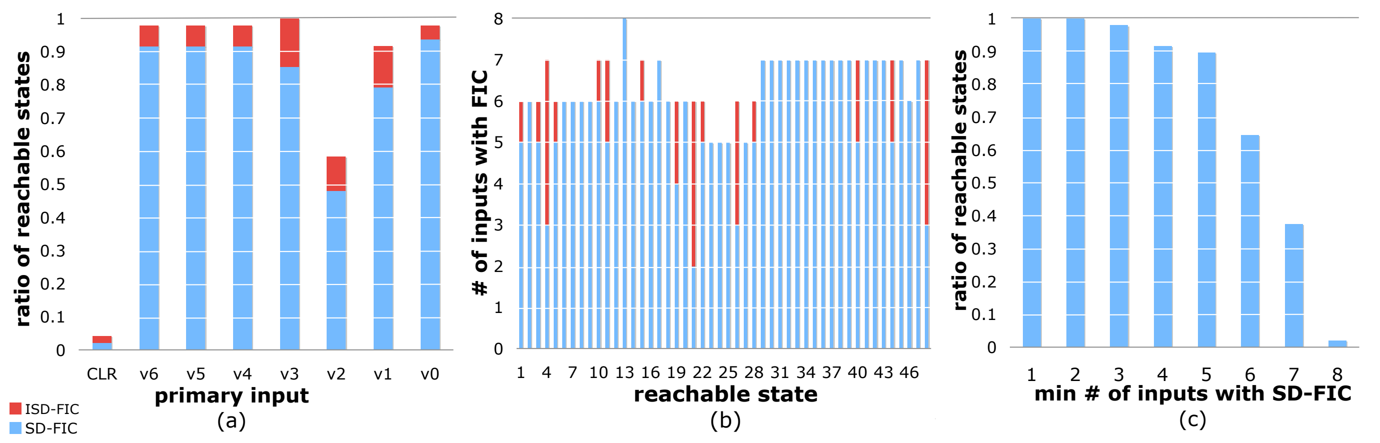


Fig. 11. Frequency distributions of functional independence conditions in s1488. In Figure 11 and 12, the acronym "ISD-FIC" refers to FIC that are functions of both state variables and at least one input variable while "SD-FIC" refers to FIC that are functions of state variables only.

Bench	PI	PO	FF	reachable states	# of inputs with SD-FIC	states with SD-FIC inputs (%)	avg. # of inputs with SD-FIC per state	# of inputs with ISD-FIC	states with ISD-FIC inputs (%)	avg. # of inputs with ISD-FIC per state
s1488	8	19	6	48	8	48 (100)	5.83	8	48 (100)	6.46
s208	10	1	8	256	8	256 (100)	7.00	9	256 (100)	9.00
s27	4	1	3	6	2	4 (66)	1.17	4	6 (100)	2.83
s298	3	6	14	218	0	0 (0)	0.00	3	218 (100)	2.06
s349	9	11	15	2625	8	2368 (90)	7.22	8	2368 (90)	7.22
s382	3	6	21	8865	0	0 (0)	0.00	3	8865 (100)	2.00
s386	7	7	6	13	5	13 (100)	4.08	7	13 (100)	6.77
s510	19	7	6	47	19	47 (100)	18.40	19	47 (100)	18.51
s526n	3	6	21	8868	0	0 (0)	0.00	3	8868 (100)	2.00
s832	18	19	5	25	17	25 (100)	14.16	18	25 (100)	16.72
s953	16	23	29	504	13	504 (100)	6.57	15	504 (100)	13.66
ex1	9	19	5	20	8	20 (100)	5.20	9	20 (100)	7.40
keyb	7	2	5	19	7	16 (84)	3.21	7	19 (100)	6.79
kirkman	12	6	4	16	6	9 (56)	2.38	11	16 (100)	9.94
planet1	7	19	6	48	7	48 (100)	5.71	7	48 (100)	6.33
sand	11	9	5	32	10	32 (100)	8.69	11	32 (100)	10.06
shiftreg	1	1	3	8	0	0 (0)	0.00	0	0 (0)	0.00
Add256Cntrl	1	2	12	24	1	23 (95)	0.96	1	23 (95)	0.96
TagGen	4	9	24	20161	0	0 (0)	0.00	2	20161 (100)	2.00
TagGenCntrl	2	2	13	23	2	22 (95)	1.87	2	23 (100)	1.91
boltzmann	7	21	93	903	6	903 (100)	5.77	6	903 (100)	5.86
lan	10	8	20	24	10	24 (100)	6.50	10	24 (100)	9.83
Avg.	7	9	14	1943	6	198 (72)	4.76	7	1931 (94)	6.74

Fig. 12. Statistics on the occurrence frequencies of functional independence conditions across all benchmarks.

Figure 12 shows the occurrence frequencies of FIC across all benchmarks. For each benchmark, column "PI", "PO", "FF" report the number of primary inputs, primary outputs, and flip-flops respectively; column "# of inputs with SD-FIC" reports the number of inputs which have satisfied SD-FIC in at least one reachable state, while column "states with SD-FIC inputs" reports the number of reachable states in which at least one input has one satisfied SD-FIC. The non-weighted average of inputs with satisfied SD-FIC per reachable states is given in the following column. The analogous analysis is applied to ISD-FIC, and results are listed in the last three columns.

These experimental results indicate that FIC are frequent in reachable states. While by definition the set of ISD-FIC includes the set of SD-FIC, the number of SD-FIC is high in most designs. In particular, all of FIC discovered in the benchmark circuit *s349* are SD-FIC.

These results confirm also that *in practice it is sufficient to focus on exploiting SD-FIC* since they already offer many opportunities to improve the performance of a latency-insensitive system. Further, the SD-FIC-detect logic is typically faster and much smaller.

B. Latency-Insensitive Design of an SoC for Wireless Communication

In the second set of experiments, we applied latency-insensitive design and the proposed FIC optimization to the semi-custom design of a system-on-chip for wireless communication in order to measure the performance improvements made possible by the FIC optimization and assess the associated overhead in terms of both area and delay.

We started from the original RTL specification of the SoC that was designed by Liu *et al.* and presented in [15], [16]: this is a "coded orthogonal frequency division modulation" (COFDM) baseband solution for ultra-wideband systems. Figure 13 shows the top-level diagram of the system: the transmitter receives packets from the medium access control (MAC) layer, and outputs encoded symbols to a DAC for physical transmission.

To evaluate the FIC optimization we actually synthesized three versions of this SoC: (1) the original or "strict" system,

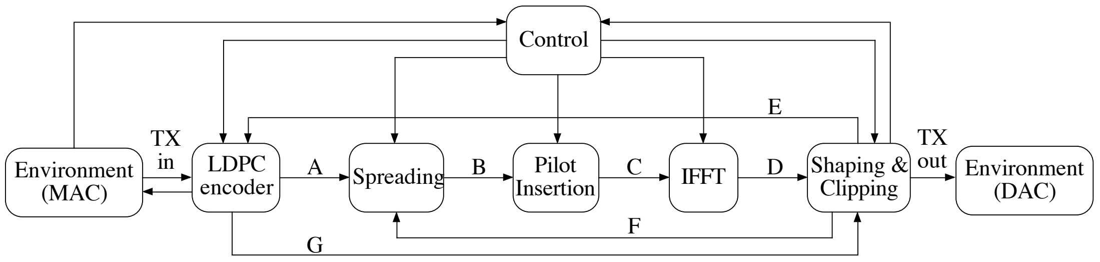


Fig. 13. The LDPC-COFDM-based ultra wideband transmitter. The channels of the datapath are labeled alphabetically.

RS	throughput		speedup	A's SD-FIC		B's SD-FIC		D's SD-FIC		E's SD-FIC		F's SD-FIC	
locations	No FIC	FIC	(%)	occurred	used	occurred	used	occurred	used	occurred	used	occurred	used
A	0.833	0.918	10.2	0.004	0.004	0.230	0.165	0.369	0.368	0.016	0.000	0.985	0.000
B	0.800	0.917	14.6	0.004	0.000	0.230	0.164	0.369	0.368	0.016	0.000	0.986	0.093
C	0.800	0.868	8.5	0.004	0.000	0.230	0.000	0.369	0.368	0.016	0.004	0.986	0.154
D	0.750	0.831	10.8	0.004	0.000	0.230	0.000	0.369	0.369	0.016	0.005	0.986	0.206
E	0.667	0.670	0.4	0.004	0.004	0.230	0.107	0.369	0.000	0.016	0.016	0.986	0.002
F	0.800	0.987	23.4	0.004	0.000	0.230	0.000	0.369	0.000	0.016	0.000	0.986	0.944
G	0.667	0.670	0.4	0.004	0.000	0.230	0.000	0.368	0.000	0.016	0.016	0.986	0.492

Fig. 14. Throughput improvements with one RS insertion and shell queues of size two.

(2) a latency-insensitive design (LID) version of it, and (3) a LID version with FIC optimization (the FIC-shell does not use the FIC-queue). We made the entire system latency-insensitive by encapsulating the five datapath modules and the controller with classic LID shells. In the third version we used the new FIC-shells, whenever applicable, [7] by exploiting the SD-FIC which are derived as explained in Section IV. These conditions are found and detected on five global communication channels (A, B, D, E, and F) that connect the datapath modules. The functional validation and throughput measurements of the two latency-insensitive systems are done by simulating the synthesizable RTL design. All of the simulations test the transmission of ten consecutive data packets, which requires more than forty thousand clock cycles. To measure the area and delay, we (a) synthesized the three designs using Synopsys Design Compiler, (b) completed technology mapping with a $90nm$ industrial standard cell library, and (c) performed static timing analysis on the mapped design.

Figure 15 reports the throughput improvements due to FIC optimization for different design configurations of the latency-insensitive SoC. The various configurations are latency-equivalent systems that differ only for the number and location of the relay stations across the seven global communication channels. All of the shells use input queues of size two. System throughput is improved in many cases and in some cases very significantly: e.g., when one or two relay stations are inserted on channel F, the FIC optimization brings the throughput almost up to 1, the ideal value. Overall the throughput speedups across all configurations range from 0.3% to 30.7% with average equal to 10.3%.

Effectiveness of FIC. All of the FIC are computed automatically without human interventions, and all but one module have at least one input channel with FIC (more precisely, SD-FIC). Some of the FIC that the tool discovered are surprisingly effective. For example, the feedback channel F from the

[7]Modules with no SD-FIC are encapsulated with classic shells. This is possible because our proposed FIC-shell follows the same LI protocol as classic shells.

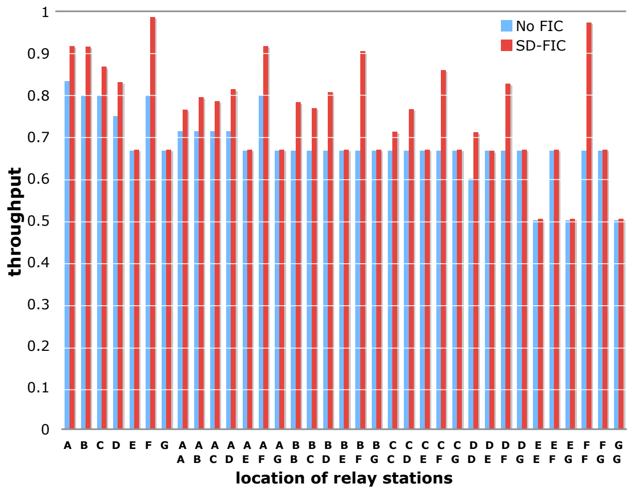


Fig. 15. Throughput improvements with one or two RS insertions on different channels.

Shaping module to the *Spreading* module is only needed in a very few number of clock cycles. Similarly, the *Pilot-Insertion* module does not need its input from channel B periodically, and this FIC often contributes to the throughput improvement.

The effectiveness of a FIC roughly depends on how often it can be used to avoid stalling of modules in the critical loops. Figure 14 reports the throughput improvements due to FIC optimization ("throughput" and "speedup" columns) for various configurations with one relay-station insertion, the frequency of the occurrences of the corresponding FIC, and the frequency of its usage to avoid stalls in the remaining columns. For example, when a relay station is inserted on channel D, the throughput is improved from 0.75 to 0.83, because the FIC of channel D and F avoid a significant number of stalls of the *Shaping* module and the *Pilot-Insertion* module respectively, and B-C-*D*-*F*-B forms the critical loop of the design. In contrast, when a relay station is inserted on channel E, the throughput remains almost the same after FIC optimization, even if B's FIC is used for stall avoidance 10% of the overall

RS	queue size = 1 throughput		queue size = 2 throughput	
locations	No FIC	FIC	No FIC	FIC
A	0.750	0.751	0.833	0.918
B	0.750	0.791	0.800	0.917
C	0.750	0.750	0.800	0.868
D	0.750	0.831	0.750	0.831
E	0.667	0.670	0.667	0.670
F	0.750	0.987	0.800	0.987
G	0.667	0.670	0.667	0.670

Fig. 16. Impact of the FIC optimization and queue sizing on the throughput (with one RS insertion).

simulation time and E's FIC is used whenever possible. This is because channel B is not on the critical loop (which is G-E-G in this case), and channel E's FIC happens rarely and thus cannot have a sizable impact on throughput.

Area and Delay Overheads. We compared the area and delay of the synthesized original transmitter versus its latency-insensitive versions with and without FIC optimization. The area overhead is minimal: 1.04% for shells with queue size of 1, and 3.26% for shells with queue size of 2. FIC-shells with FIC-detect blocks add negligible area (less than 0.01%) to the classical shells. The critical-path delay of both the classic and FIC-optimized latency-insensitive transmitters are the same as in the original strict design, i.e. the maximum clock speed is not affected. Consequently, FIC optimization often increases the latency-insensitive system's *effective performance*, which is defined as clock frequency times system throughput [7].

FIC Optimization and Queue Sizing. As discussed in [8], [17], [18], the input queue sizes in the shells also affect system throughput. This is because reconvergence paths with different end-to-end latencies caused by relay-station insertions can become a critical *loop* consisting of forward data paths and backward backpressure paths. For example, if we insert one relay station on channel F in an LID implementation of our design where queues have size one, the reconvergence paths E-A-F becomes a critical loop with a cycle mean of $4/3$ (so the throughput is $3/4 = 0.75$). In order to avoid this throughput degradation, one option is to increase the size of the shell's input queues. For instance, to increase the queue sizes to two makes it possible to raise the throughput to 0.8. Columns labeled as "No FIC" in Figure 16 report analogous throughput variations due to different queue sizes when the relay station is inserted on one of the global channels.

On the other hand, the use of FIC creates more opportunities for throughput optimization. For instance, instead of sizing the queues, we can exploit a FIC of channel F to bring the throughput back to 0.98. This requires less area overhead and achieves higher throughput than the queue sizing technique. In other scenarios, e.g. if the relay station is inserted on channel B, to combine queue sizing and FIC optimization can achieve a higher throughput (0.92) than using only one technique alone (0.80 for queue sizing only or 0.79 for FIC optimization only). Columns labeled as "FIC" in Figure 16 report the throughput data for the various scenarios.

Sizing of Shift Registers. The sizes of the shift registers affect the achievable throughput optimizations through FIC. Intuitively the larger the shift registers are, the more opportunities to exploit FIC for throughput optimizations by *consecutive* stall avoidance. When a shift register is full, the FIC-shell can no longer exploit FIC of the corresponding input channel.

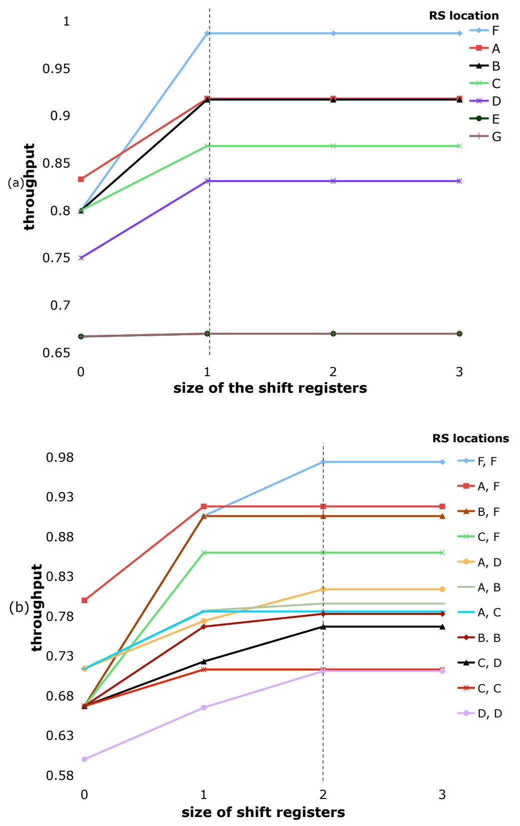


Fig. 17. Impact of the size of the shift registers in the FIC-shells on system throughput for the latency-insensitive design of the COFDM: (a) One RS insertion; (b) Two RS insertions.

On the other hand, the throughput improvements by enlarging shift registers are also limited, since the location of the relay stations and the timing of FIC occurrences are the inherent deciding factors [8]. Figure 17 reports the impacts of the size of the shift registers on system throughput. Figure 17(a) measures the throughput of the latency-insensitive COFDM design with one RS insertion on one of the seven global data channels; Figure 17(b) measures the throughput of the same design but with two RS insertions. In both sets of experiments, the sizes of all the shift registers vary from zero to three across all FIC-shells. Note that a FIC-shell using shift registers of size zero "degenerates" to a classic shell, so the leftmost data points in the two figures are the throughput values of the systems using classic LID shells. Also, note that in the case of inserting one relay station, the throughput improvements stop after the size of the shift registers becomes larger than one. Similarly, in the second set of experiments of inserting two relay stations, the throughput levels off when the size of the shift registers exceeds two. This shows that for the latency-insensitive COFDM design small shift registers suffice to enable all of the possible FIC optimization opportunities.

Evaluations of FIC-queue. We also compared the FIC optimization using the FIC-queue technique to the one without using it, whose results are presented earlier. We found that the

[8]Assuming the environment does not constrain the system throughput.

Bench	PI	PO	FF	reachable states	# of inputs with SD-FIC	states with SD-FIC inputs (%)	avg. # of inputs with SD-FIC per state	# of inputs with ISD-FIC	states with ISD-FIC inputs (%)	avg. # of inputs with ISD-FIC per state
s1488	8	19	6	48	8	48 (100)	5.83	8	48 (100)	6.46
s208	10	1	8	256	8	256 (100)	7.00	9	256 (100)	9.00
s298	3	6	8	135	2	7 (5)	0.10	3	135 (100)	2.09
s27	4	1	3	5	2	3 (60)	1.00	4	5 (100)	3.40
s349	-	-	-	-	-	- (-)	-	-	- (-)	-
s382	-	-	-	-	-	- (-)	-	-	- (-)	-
s386	7	7	4	13	5	13 (100)	4.08	7	13 (100)	6.77
s510	19	7	6	47	19	47 (100)	18.40	19	47 (100)	18.51
s526n	-	-	-	-	-	- (-)	-	-	- (-)	-
s832	18	19	5	24	17	24 (100)	14.08	18	24 (100)	16.67
s953	-	-	-	-	-	- (-)	-	-	- (-)	-
ex1	9	19	5	18	8	18 (100)	5.28	9	18 (100)	7.39
keyb	7	2	5	19	7	16 (84)	3.21	7	19 (100)	6.79
kirkman	12	6	4	16	6	9 (56)	2.38	11	16 (100)	9.94
planet1	7	19	6	48	7	48 (100)	5.71	7	48 (100)	6.33
sand	11	9	5	32	10	32 (100)	8.69	11	32 (100)	10.06
shiftreg	1	1	3	8	0	0 (0)	0.00	0	0 (0)	0.00
Add256Cntrl	1	2	5	18	1	17 (94)	0.94	1	17 (94)	0.94
TagGen	-	-	-	-	-	- (-)	-	-	- (-)	-
TagGenCntrl	2	2	5	19	2	18 (94)	1.84	2	19 (100)	1.89
lan	10	8	5	23	10	23 (100)	6.52	10	23 (100)	9.83
Avg.	8	8	5	48	7	38 (82)	5.60	8	47 (92)	7.51

Fig. 18. Statistics on the occurrence frequencies of functional independence conditions across all benchmarks subjected to state minimization, sequential and combinational logic optimizations. Benchmarks whose state space cannot be handled by the tool are marked by dashes in the corresponding rows.

throughput improvements of using FIC-queue on the COFDM design are few. The only throughput improvement is seen in the case of inserting one relay station on both channel A and channel E. The real storage space of queues in each shell is one. The throughput of the design increase from 0.60 to 0.66. Applying FIC-queue to other design configurations, the throughput remains the same.

C. Discussion of FIC-Based Optimization

The presence of functional independence conditions is mostly due to the *behavior* of a design, not to the sub-optimality of the implementation of its logic circuits. That is, the behavior of a core module or the entire system *implicitly* introduces the FIC. Hence, when the core is implemented as a netlist of logic gates, our algorithm automatically constructs FIC based on the logic structure by operating at the circuit level. This claim is supported by the analysis of the experimental results for those cases where the behavior of the design is known:

1) Benchmark *s1488*, whose FIC are analyzed in Figure 11, is an add-shift-multiplier [19] controlled by a 3-bit counter. *By design*. its inputs are only needed in the first cycle of each round of multiplication. This explains why this benchmark has many state-dependent FIC.
2) For the case of the COFDM SoC the occurrence of FIC of channel B may be traced back to the specification of the standard protocol as given in [20]: the *Pilot-Insertion* module adds pilot symbols periodically to allow a receiver to measure the distortions of the transmitted symbols and when it operates in this mode it does not need the inputs from channel B.

A second observation is that *logic optimizations do not affect the amount of FIC discovered by our algorithm*. We repeat the same analysis as presented in Figure 12 measuring the occurrence frequencies of FIC for the same suite of benchmarks after applying state minimization with STAMINA [21] and the logic optimization scripts in ABC [9]. The corresponding results are presented in Figure 18 [10]. The comparing of the two sets of results presented in Figure 12 and in 18 shows that the frequency of FIC occurrence frequencies are almost the same. This means that the optimization of the logic structures does not significantly affect the number of FIC.

Also, while the synthesis of the COFDM design that is returned by Synopsys Design Compiler includes also various logic optimization steps, our procedure still identifies FIC that are induced by the COFDM communication protocol. In fact, this should not be a surprise if one accepts that FIC depend on the functional specification (the behavior) of the design, which is not changed by a logic synthesis tool.

As a final note, we would like to stress the ability of the proposed algorithm to discover the FIC automatically regardless of the nature of the design and without human interventions. For example, our method discovers the FIC in the COFDM design automatically without the knowledge of its logic design and protocol design, and synthesizes the necessary FIC-detection logic in a "correct-by-construction" fashion.

VI. RELATED WORK

FIC-based optimization is related to the concept of *early evaluation* in *asynchronous* circuit and system design. Early evaluation allows an asynchronous component to compute its output before all of its input values are available. It is a more practical restriction of the OR-causality precedence relation for which Yakovlev *et al.* provide formal models and implementations for speed-independent asynchronous circuits in [22], [23]. Early evaluation has been applied to phased logic at different granularity levels by Reese *et al.* [24], [25] and to the optimization of pipelined asynchronous logic by both Brej *et al.* [26] and, more recently, Ampalam and Singh [27].

[9]For the original analysis we did not apply any sequential/combinational logic optimizations to the benchmarks.

[10]The state space of certain benchmarks cannot be handled by STAMINA.

Early evaluation can be extended to synchronous circuits if these operate according to a latency-insensitive protocol. The idea has been first investigated in the context of multi-clock latency-insensitive circuits in [28], [29], and it has been applied to elastic systems by using a new latency-insensitive protocol that explicitly encodes anti-token signals [30]. Both the work by Casu and Macchiarulo on *adaptive* latency-insensitive protocols [13] and our preliminary results on FIC-based optimization [31] have shown that unnecessary stalling can be avoided with local modification in the logic design of a shell and without requiring any change in the channel interface signals (void and stop bits) that were defined to implement the original latency-insensitive protocol [9].

Two ingredients common to early evaluation and FIC-based optimization are: the design of the detection logic and the mechanism to implement delayed stalls for dealing with late-arriving, previously-unneeded data items (see Section II-A).

Detection Logic. To improve performance with early evaluation or exploiting FIC, a mechanism to dynamically detect the occurrence of such event must be supplied. Most approaches in the literatures assume that this functionality has to be manually designed. The burden of manual design is partially reduced in the method described in [29], which however requires designers to provide high-level specifications of triggering functions that are then automatically translated into FSM implementations.

Casu and Macchiarulo identify the need to have an "effective and simple" combinational logic block, which they call "oracle", to implement the detection logic, but they do not provide a method to synthesize it [13]. All the aforementioned approaches somewhat assume that the designers have full knowledge of the triggering conditions. Instead, the notion of FIC and the logic synthesis procedure for the FIC-detect logic block that we have presented in Section IV establish an automatic solution for this problem that does not request any effort from the designers. Such automatic procedures are possible because an implementation of the functional specification of a core contains all the necessary information. Fully-automatic synthesis approaches are obviously more desirable since they eliminate human errors and simplify the application of the proposed optimization method.

Reese *et al.* in [24] provide an algorithm based on traversing root-to-terminal paths in a BDD representing the given logic function. This method applies to the synthesis of one trigger function on a fixed subset of inputs. Our procedure, which uses unobservability conditions, targets arbitrary multi-input and multi-output logic functions and finds all the triggering conditions on all of the possible input subsets.

Handling Delayed Stalls. One challenge of both early evaluation and FIC-based optimization is to ensure the functional correctness of the final implementation. If a logic component evaluates its outputs in the absence of a valid data token, when the absent valid token finally arrives it will be obsolete and, therefore, unusable for correct computation. Hence, it is necessary to ensure that all the computations are fired on the fresh data tokens. To deal with this problem, various approaches have been proposed that are either based on asynchronous design styles or assume various kinds of global synchronization schemes, among which are synchronous latency-insensitive systems. Still, even though these methods apply to distinct design styles, they can be divided into three broad classes.

One class of methods assumes communication protocols which use *explicit acknowledgement* to request new wave of data tokens as in many asynchronous systems. The idea is to withhold the acknowledgement until all data arrive, even if some early arrivals already trigger the computation. Reese *et al.* use Petri nets to model and implement such a handshaking mechanism for asynchronous phased-logic systems [24], [25].

An alternative approach is to augment the communication infrastructure with flow of *anti-tokens*, which run in parallel with the normal data flow but in the opposite direction and annihilate unused (and unneeded) normal data tokens [26], [27], [30]. Whenever a computation core early evaluates, it generates one anti-token for each input channel from which a late token is expected. Such mechanisms require communication protocols that accommodate the flow of anti-tokens as well as carefully-designed interface circuits which propagate and destroy normal tokens and anti-tokens properly.

The third approach is based on *counting* the number of subsequent tokens to be discarded due to early evaluations for each input channel. This notion is similar to the accumulation of negative tokens in the "guarded" Petri net model proposed by Júlvez *et al.* for performance analysis of early evaluation [32]. Casu and Macchiarulo [13] implement this technique by using an up-down counter for each input channel whose value is the number of tokens to be discarded. We use a 1-bit shift register instead of an up-down counter to reduce the hardware overhead [11].

Compared to the anti-token and counting-based approaches, the explicit acknowledgement method is more restrictive. The withholding of the acknowledgements is equivalent to increasing the counter value to one, but it also prohibits "consecutive" early firings, which result in greater counter values if a counting-based approach is used. Thus, the acknowledgement method loses some optimization opportunities that are possible with the other two techniques: the counting-based approaches support back-to-back consecutive early firings by allowing greater-than-one counter values; the anti-token techniques achieves the same effect by sending out anti-tokens continuously as long as there is no traffic congestion of the anti-token flows. Interestingly, compared to using anti-tokens, the counting-based method can be viewed as storing (the number of) the anti-tokens locally in queues, which provide buffering mechanism. Finally, while the communication interfaces supporting anti-token flows require the modification of the global communication protocols, the counting-based methods, including ours, do not as they only demand changes that are inherently "local" to the interface.

[11]Casu and Macchiarulo have proposed a novel technique to use FIC not only to reduce the number of stalls caused by void tokens but also the stalls caused by backpressures. This is achieved by discarding valid but not needed data tokens which cannot be immediately used, instead of requesting its sender to repeat sending the same data. In such cases the counter value is decreased from zero to *negative one*, in order to properly align the next wave of data tokens. In Section III-C we showed how this idea can be generalized to virtually increase the queue sizes in our FIC-shells for backpressure reduction.

VII. Conclusions

We studied the problem of leveraging the local knowledge on the internal logic of a core to improve the global SoC performance in latency-insensitive design. We defined the notion of functional independence conditions (FIC) and we described a logic synthesis procedure to generate automatically a shell interface (a FIC-shell) around a given a core that dynamically detects FIC occurrences to avoid unnecessary local stalling of the core, thereby increasing the overall system performance. We presented a comprehensive experimental study that includes: an evaluation of the applicability and practicality of the proposed technique with a suite of benchmark circuits and the complete semi-custom design of an SoC for wireless communication. Experimental results show that on average the data processing throughput of this SoC can be increased by up to 30% with an area overhead that is never larger than 3.26%.

Acknowledgments

The authors gratefully thank Hsuan-Yu Liu and Chen-Yi Lee for providing the RTL design of the COFDM SoC and Claudio Pinello for useful discussions on FIC-based optimization.

References

[1] 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.

[2] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SOC Revolution: A Guide to Platform Based Design*. Kluwer Academic Publishers, 1999.

[3] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwer Academic Publishers, 1998.

[4] L. Scheffer, "Methodologies and tools for pipelined on-chip interconnect," in *Proceedings International Conference on Computer Design*, Oct. 2002, pp. 152–157.

[5] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, December 2000.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[7] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proceedings of the Design Automation Conference*, Jun. 2000, pp. 361–367.

[8] R. Lu and C.-K. Koh, "Performance analysis of latency-insensitive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 469–483, Mar. 2006.

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

[10] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, ser. Electrical and Computer Engineering Series. McGraw-Hill Book Company, 1994.

[11] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 6, pp. 723–740, 1988.

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

[13] M. R. Casu and L. Macchiarulo, "Adaptive latency-insensitive protocols," *IEEE Design and Test of Computers*, vol. 24, no. 5, pp. 442–452, Sept.-Oct. 2007.

[14] "ABC: A system for sequential synthesis and verification," [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/.

[15] H.-Y. Liu, C.-C. Lin, Y.-W. Lin, C.-C. Chung, K.-L. Lin, W.-C. Chang, L.-H. Chen, H.-C. Chang, and C.-Y. Lee, "A 480mb/s LDPC-COFDM-based UWB baseband transceiver," in *ISSCC Digest of Technical Papers*, vol. 1, 2005, pp. 444–609.

[16] C.-Y. Lee, H.-Y. Liu, and C.-C. Lin, "SoC for COFDM wireless communications: Challenges and opportunities," in *International Symposium on VLSI Design, Automation and Test*, 2006, pp. 1–4.

[17] R. Collins and L. P. Carloni, "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system," in *Proceedings of the Design Automation Conference*, Jun. 2007, pp. 410–416.

[18] R. Lu and C.-K. Koh, "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels," in *Proceedings International Conference on Computer-Aided Design*, 2003, pp. 227–231.

[19] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering," *IEEE Design and Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[20] A. Batra, J. Balakrishnan, A. Dabak, R. Gharpurey, P. Fontaine, J. Lin, J.-M. Ho, S. Lee, M. Frechette, S. March, and H. Yamaguchi, "Multi-band OFDM physical layer proposal for IEEE 802.15 task group 3a," *IEEE P802.15-03/268r1-TG3a*, Sep. 2003.

[21] J.-K. Rho, G. D. Hachtel, F. Somenzi, , and R. M. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 2, pp. 167–177, Feb. 1994.

[22] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny, "On the models for asynchronous circuit behaviour with OR causality," *Journal of Formal Methods in System Design*, vol. 9, no. 3, pp. 189–233, 1996.

[23] A. Bystrov, D. Sokolov, and A. Yakovlev, "Low-latency control structures with slack," in *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, May 2003, pp. 164–173.

[24] R. R. Reese, M. A. Thornton, and C. Traver, "A coarse-grain phased logic CPU," in *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2003, pp. 2–13.

[25] R. R. Reese, M. A. Thornton, C. Traver, and D. Hemmendinger, "Early evaluation for performance enhancement in phased logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, pp. 532–550, Apr. 2005.

[26] C. F. Brej and J. D. Garside, "Early output logic using anti-tokens," in *Proceedings International Workshop on Logic Synthesis*, 2003, pp. 302–309.

[27] M. Ampalam and M. Singh, "Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens," in *Proceedings International Conference on Computer-Aided Design*, 2006, pp. 611–618.

[28] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2004, pp. 1008–1013.

[29] A. Agiwal and M. Singh, "An architecture and a wrapper synthesis approach for multi-clock latency-insensitive systems," in *Proceedings International Conference on Computer-Aided Design*, 2005, pp. 1006–1013.

[30] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *Proceedings of the Design Automation Conference*, 2007, pp. 416–419.

[31] C.-H. Li and L. P. Carloni, "Using functional independence conditions to optimize the performance of latency-insensitive systems," in *Proceedings International Conference on Computer-Aided Design*, 2007, pp. 32–39.

[32] J. Júlvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Proceedings International Conference on Computer-Aided Design*, 2006, pp. 448–455.