Leveraging Local Intracore 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 intracore logic design from the design of the intercore 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 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 downlink 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 condition (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 intracore 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 semicustom design of an SoC, an ultrawideband baseband transmitter, using a 90-nm industrial standard cell library.**

***Index Terms*—Finite state machines (FSMs), latency-insensitive design (LID), logic synthesis, sequential logic optimization, system-level design, system-on-chip (SoCs).**

I. Introduction

DESIGNERS of systems-on-chip (SoCs) 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 semicustom 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 predesigned and prevalidated *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*, a technique to fix timing violations in global interconnect that is very effective yet challenging to apply [4].

Manuscript received January 27, 2008; revised July 21, 2008. Current version published January 21, 2009. This work was supported in part by Intel Corporation and in part by the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This paper was recommended by Associate Editor V. Narayanan.

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

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2009157

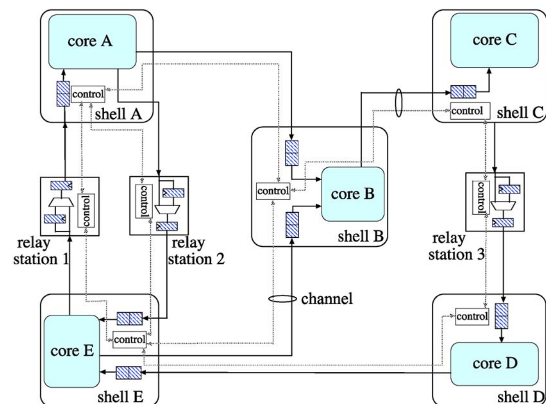


Fig. 1. Shell encapsulation, relay-station insertion, and channel backpressure.

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 is 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.

Fig. 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, etc.) as long as it satisfies the requirement that it is *stallable*, i.e., it can be *clock gated*. The shell dynamically controls

0278-0070/$25.00 © 2009 IEEE

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

the operations of the core by deciding whether to stall 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 are 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 twofold storage capacity, unit latency, 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 any variations of delay on *intercore* 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 *intracore* logic design [6].

LID helps to meet the required target clock frequency 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 Fig. 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 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 this 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* can exploit dynamically its core's functional independence condition (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. For instance, this may occur for 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 clock cycle, FIC depend on the *local* logic state of the core and, potentially, on a subset of the data on 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.

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 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 semicustom 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 an FSM. We revisit in the following the classic FSM model in the context of LID to highlight the role played by the core's I/O channels (Fig. 2).

1) The inputs of the FSM is a set of Boolean variables, partitioned into N groups: $P = P_1 \bigcup \cdots \bigcup P_N$, where P_i is a set of w_i Boolean variables $\{p_1^i, \ldots, p_{w_i}^i\}$, representing the data portion of an input channel i of parallelism w_i.
2) The outputs of the FSM is a set of Boolean variables partitioned into M groups: $Q = Q_1 \bigcup \cdots \bigcup Q_M$, where $Q_j = \{q_1^j, \ldots, q_{w_j}^j\}$ represents the data of an output channel j of parallelism w_j.
3) 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 become the present state's values.

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

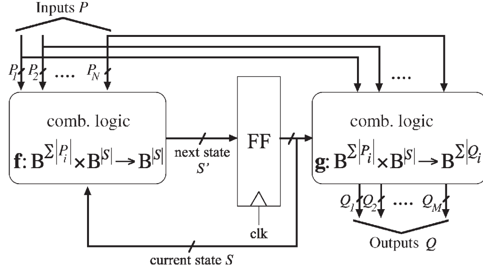


Fig. 2. Modeling a core module as an FSM.

4) 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 $f_i : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}$, or simply $\mathbf{f} : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}^{|S|}$. 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 just $\mathbf{g} : \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathsf{B}^{|S|} \rightarrow \mathsf{B}^{|Q_1|+\cdots+|Q_M|}$.

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 an 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 $\mathrm{T}' \equiv \{\widetilde{\mathrm{P}_1}, \ldots, \widetilde{\mathrm{P}'_k}, \ldots, \widetilde{\mathrm{P}_N}; \widetilde{S}\}$ that only differs for the value of input channel P_k, we have

$$\mathbf{f}(\mathrm{T}) = \mathbf{f}(\mathrm{T}') \quad (1)$$

$$\mathbf{g}(\mathrm{T}) = \mathbf{g}(\mathrm{T}'). \quad (2)$$

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]

$$FIC_{P_k}(\mathrm{T}) \stackrel{\text{def}}{=} \{\widetilde{\mathrm{P}}_1, \ldots, \widetilde{\mathrm{P}}_{k-1}, \widetilde{\mathrm{P}}_{k+1}, \ldots, \widetilde{\mathrm{P}_N}; \widetilde{\mathrm{S}}\}$$

a ***FIC*** of input channel P_k. If either $\mathbf{f}$ or $\mathbf{g}$ is dependent on the value of P_k, $FIC_{P_k}(\mathrm{T}) \stackrel{\text{def}}{=} \emptyset$.

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 set of FIC of channel P_k is

$$FIC_{P_k} \stackrel{\text{def}}{=} \bigcup_{\mathrm{T} \in \mathcal{T}} FIC_{P_k}(\mathrm{T}). \quad (3)$$

Since the number of distinct input and present-state values is finite, the set FIC_{P_k} is also finite. The set of FIC_{P_k} can be partitioned in the following two subsets.

1) $SDFIC_{P_k}$ is the subset of FIC_{P_k} which depends only on the present state, i.e., $SDFIC_{P_k} = \bigcup_{\mathrm{T} \in \mathcal{W}} FIC_{P_k}(\mathrm{T})$, where $\mathcal{W} = \mathsf{B}^{|P_1|+\cdots+|P_N|} \times \mathcal{S}$, with $\mathcal{S} = \{\widetilde{\mathrm{S}} | \forall \widetilde{\mathrm{P}_1}, \ldots, \forall \widetilde{\mathrm{P}_N}, FIC_{P_k}(\{\widetilde{\mathrm{P}_1}, \ldots, \widetilde{\mathrm{P}_n}; \widetilde{\mathrm{S}}\}) \in FIC_{P_k}\}$.
2) $ISDFIC_{P_k}$ is the subset of FIC_{P_k} which depends both on input and present-state values, i.e., $ISDFIC_{P_k} = FIC_{P_k} \setminus SDFIC_{P_k}$.

[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].

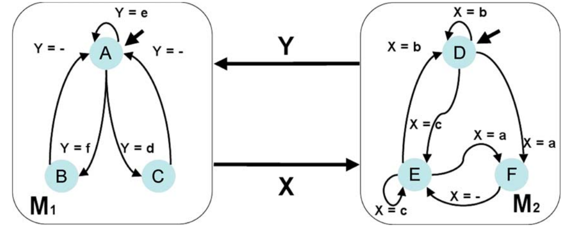


Fig. 3. Synchronous system made up of two communicating FSMs.

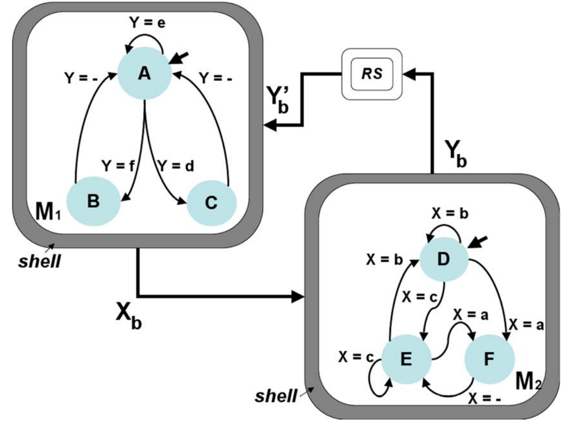


Fig. 4. Latency-insensitive system derived from the system of Fig. 3.

Next, we present a simple example to show how FIC can be used to optimize the performance of a latency-insensitive system. 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 Fig. 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 being in state B, and $X = c$ while being in state C. Similarly, FSM M_2 outputs $Y = d$ while being in state D, $Y = e$ while being in state E, and $Y = f$ while being in state F. As denoted by the arrow, the initial states are A for M_1 and D for M_2. There are three sets of traces in Fig. 5: The first set captures the behavior of the

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

		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. Sets of traces for the behaviors of the three systems in the motivating example.

strictly synchronous system of Fig. 3. Notice that the system cycles through five compound state transitions: For M_1, we have $(A \to C \to A \to A \to B) \to (A \to C \ldots)$, while for M_2, we have $(D \to F \to E \to F \to E) \to (D \to F \ldots)$.

The second set of traces in Fig. 5 describes the behavior of the system of Fig. 4: This latency-insensitive system is obtained from the system of Fig. 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, 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 and Y traces with the X_b and Y_b traces shows that they are *latency equivalent* as expected, i.e., they are the same if one ignores the τ symbol [6]. However, the data processing throughput of the system is reduced from 1 to $2/3 = 0.66$.

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 q under all possible input patterns: $FIC_{X_b} \equiv \{*; S_{M_2} = F\}$. This FIC can be used to design a shell that performs the following: 1) Avoids to stall M_2 whenever it is in state F and there is a τ on channel X_b (*stall avoidance*), and 2) 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 Fig. 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 $5/7 = 0.72$.

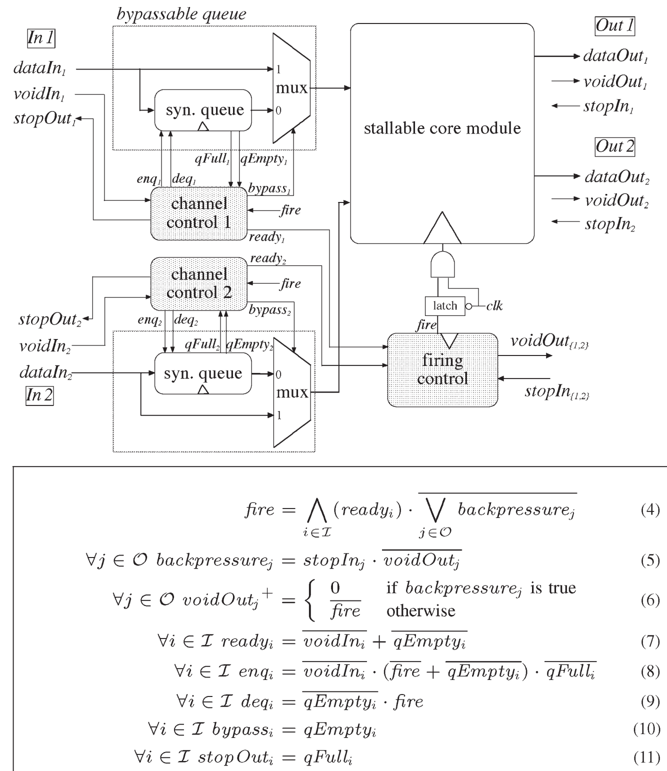


Fig. 6. (Top) Block diagram of a two-input–two-output shell around a stallable core module; (bottom) channel and firing control logic of the shell.

III. Shell Design

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

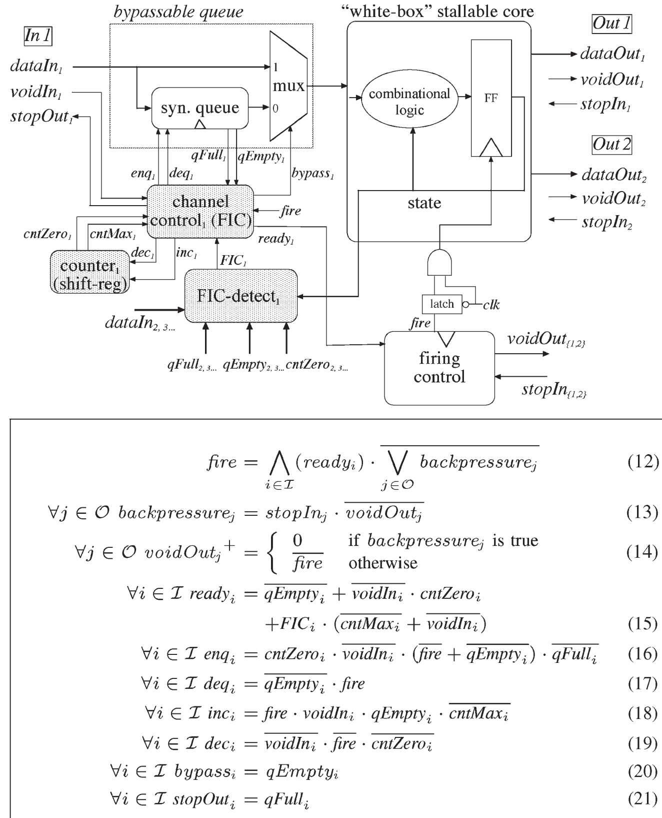
A. Classic Shell With Backpressure

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 (although 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. The top of Fig. 6 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 are 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 shown in the bottom half of Fig. 6. At each clock cycle, the shell decides whether the computation of a core module can proceed: The computation is allowed for

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

the next clock cycle ("firing") if and only if the *fire* signal is high (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 (7). The signal $backpressure_j$ is set high if a downlink receiver of an output channel j is unable to save the valid data generated by the sender in its own queue; it is indicated by (5), where $stopIn_j = 1$ and $voidOut_j = 0$. The output tokens generated by a stalled module are marked as void if there is no backpressure from the downlink receivers [the second clause in (6)]. In the case of backpressure from a receiver on an output channel j, the current valid output data will not be changed (due to stalling caused by backpressure) and will be marked as valid until the receiver has storage space to save it [the first clause in (6)]. Equations (8)–(10) are the rules for steering the input data. For an input channel i, the valid but not consumed (due to stalling) data are stored in its queue for later use, indicated by $enq_i = 1$ as in (8). If the core is fired and the queue is not empty, the data at the head of the queue are used by the core, and thus, it is dequeued [$deq_i = 1$, (9)]. The $bypass_i$ signal directs the proper data to the core, either from the channel or from the output of the queue (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 (11).



$$fire = \bigwedge_{i \in \mathcal{I}} (ready_i) \cdot \overline{\bigvee_{j \in \mathcal{O}} backpressure_j} \tag{12}$$

$$\forall j \in \mathcal{O}\ backpressure_j = stopIn_j \cdot \overline{voidOut_j} \tag{13}$$

$$\forall j \in \mathcal{O}\ voidOut_j{}^{+} = \begin{cases} 0 & \text{if } backpressure_j \text{ is true} \\ \overline{fire} & \text{otherwise} \end{cases} \tag{14}$$

$$\begin{aligned} \forall i \in \mathcal{I}\ ready_i &= \overline{qEmpty_i} + \overline{voidIn_i} \cdot cntZero_i \\ &\quad + FIC_i \cdot (\overline{cntMax_i} + \overline{voidIn_i}) \end{aligned} \tag{15}$$

$$\forall i \in \mathcal{I}\ enq_i = cntZero_i \cdot \overline{voidIn_i} \cdot (\overline{fire} + \overline{qEmpty_i}) \cdot \overline{qFull_i} \tag{16}$$

$$\forall i \in \mathcal{I}\ deq_i = \overline{qEmpty_i} \cdot fire \tag{17}$$

$$\forall i \in \mathcal{I}\ inc_i = fire \cdot voidIn_i \cdot qEmpty_i \cdot \overline{cntMax_i} \tag{18}$$

$$\forall i \in \mathcal{I}\ dec_i = \overline{voidIn_i} \cdot \overline{fire} \cdot \overline{cntZero_i} \tag{19}$$

$$\forall i \in \mathcal{I}\ bypass_i = qEmpty_i \tag{20}$$

$$\forall i \in \mathcal{I}\ stopOut_i = qFull_i \tag{21}$$

Fig. 7. (Top) Block diagram of an FIC-shell; (bottom) FIC-shell channel and firing control logic. For clarity, only input channel $i = 1$ is shown.

B. Design of an FIC-Shell

Fig. 7 shows a block diagram of the newly proposed FIC-shell design and its logic. 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 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 these data arrive, they must be discarded. Therefore, for each input channel, a FIC-shell maintains a counter that records the number of cycles that the core module currently runs ahead with respect to the next valid data on the channel.

For an input channel i, whether certain FIC for the channel is satisfied at a given clock cycle is dynamically established by the $FIC\text{-}detect_i$ 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 $FIC\text{-}detect_i$ sets FIC_i high, the current data of the channel are not needed for the core's computation. In Section IV, we present a procedure for the logic synthesis of this block.

The bottom of Fig. 7 lists the channel and firing control logic of the FIC-shell. As indicated earlier, the firing control logic [(12)–(14) shown in Fig. 7] is the same as the one of the classic shell. The channel control logic, instead, 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 (18) and (19): 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 one (18). A nonzero count indicates that the next valid data are outdated and should be discarded on arrival. If this causes a delayed stall, the count is decreased by one (19). An input channel i is ready when any of the following conditions holds (15):

1) The queue provides valid data $(qEmpty_i = 0)$.
2) The channel provides valid $(voidIn_i = 0)$ and fresh data (marked by the zero count, i.e., $cntZero_i = 1$).
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 are 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 void data and the FIC_i is true, because exploiting the

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

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

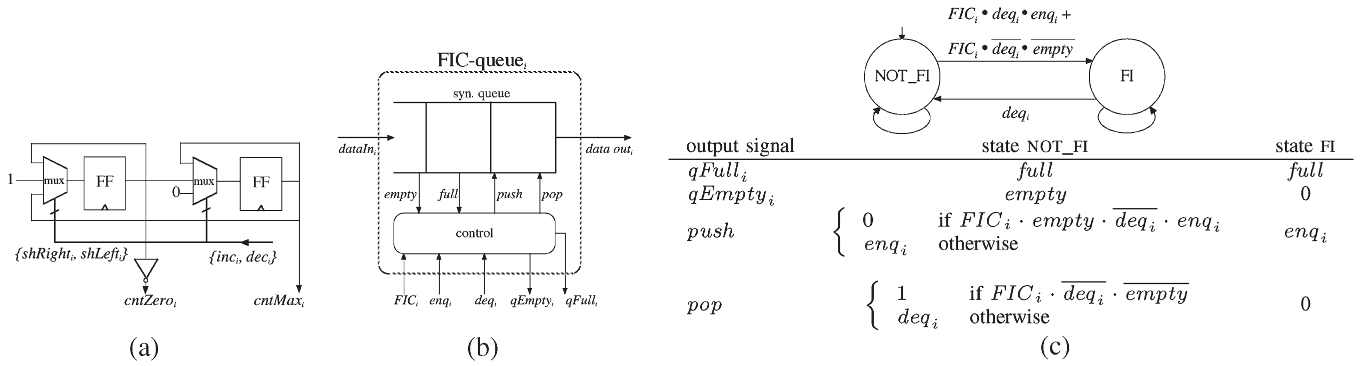


Fig. 8. Some components of a FIC-shell. (a) Shift register used as the counter. (b) Block diagram of the FIC-queue for a given input channel i. (c) FSM and output signals of the FIC-queue.

FIC for stall avoidance 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 *valid* data $(voidIn_i = 0)$ are received and the core's computation is independent from the data $(FIC_i = 1)$. These valid data are 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). Thus, regardless of the maximum value of the count, the FIC-shell can always 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. Fig. 8(a) 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 $(cntMax_i)$. The size of the shift registers affects the amount of throughput recovery that can be obtained with FIC optimization after the insertion of relay stations. Specifically, if a core presents enough FIC to avoid n consecutive stalls due to the insertion of n relay stations on a feedback loop, then the shift register must have at least n empty slots in order to avoid becoming a limiting factor for the system performance.

We use synchronous queues to save unused but valid data tokens. The enqueing and dequeuing operations only take effect at the rising/falling clock edges, i.e., the 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 queue implementation has no combinational path between its inputs and outputs.

Remark: A FIC-shell relaxes the conditions of firing a core module but still follows the latency-insensitive protocol when it communicates with relay stations or other shells. Thus, *FIC-shells and classic shells can coexist in a system.* Therefore, a designer can use FIC-shells only when it is beneficial to the system's performance (like for cores in the critical feedback loops [7], [8]), while classic shells are sufficient elsewhere.

C. FIC-Queue: Reducing the Stalls Due to Backpressure

FIC can also be used to "virtually" increase the queue sizes without allocating real storage elements for data to reduce the stalls due to backpressure. 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 are 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.

Fig. 8(b) shows the block diagram of the queue, its internal signals, and its external interface with the remaining logic of the shell. The FIC-queue replaces the original queue in Fig. 7 and subsumes its functionality. 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 unused 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. Fig. 8(c) 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 data not needed for the core's computation exist, but their values are discarded. The control discards data not critical to the computation of the core and enters the FI state in either of the following two scenarios.

1) If the queue is empty, the core's computation does not depend on the input data $(FIC_i = 1)$, and if 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 signals as they

[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.

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

```
COMPUTE-FIC(core = {f, g}, P = {P_1, ..., P_N})
     ▷ f and g are core's state transition and output functions;
       P is the set of input channels.
 1   for each channel P_i ∈ P
 2       ~FIC_{P_i}(f, g) ← true
 3       for each scalar function h ∈ f ∪ g
 4           ~FIC_{P_i}(h) ← true
 5           for each Boolean input variable p_k^i ∈ P_i
 6               if p_k^i is in the support of h then
 7                   ¬(∂h/∂p_k^i) ← h|_{p_k^i=1} ⊕̄ h|_{p_k^i=0}
 8                   ~FIC_{P_i}(h) ← ~FIC_{P_i}(h) ∧ ¬(∂h/∂p_k^i)
 9                   discard ¬(∂h/∂p_k^i)
10           ~FIC_{P_i}(f, g) ← ~FIC_{P_i}(f, g) ∧ C_{P_i}(~FIC_{P_i}(h))
11       if FIC = ISDFIC ∪ SDFIC is required then
12           FIC_{P_i}(f, g) ← Replace each literal p in ~FIC_{P_i}(f, g) with
                 p · ¬(voidIn_k · cntZero_k + ¬qEmpty_k), and ¬p
                 with ¬p · (voidIn_k · cntZero_k + ¬qEmpty_k)
13       else ▷ SDFIC only
14           SDFIC_{P_i}(f, g) ← C_P(~FIC_{P_i}(f, g))
15       Save FIC_{P_i}(f, g) (or SDFIC_{P_i}(f, g)) as P_i's FIC-detect.
```

Fig. 9. Algorithm to identify FIC for FIC-detect synthesis.

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 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 as FIC-detect blocks.

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 the conditions under which f is sensitive to x. The Boolean difference is the XOR $(\oplus)$ of the cofactors of f with respect to x and $\overline{x}$. Thus, the conditions under which function f is insensitive to variable x is

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

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

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

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

B. Synthesis Algorithm

Fig. 9 shows the FIC-detect synthesis algorithm. The inputs are a core modeled by its state transition and output functions $\mathbf{f}$ and $\mathbf{g}$, and the core's input channels $\{P_1, \ldots, P_N\}$. The algorithm's main loop iteratively computes the FIC for each input channel and saves the FIC as the channel's FIC-detect logic (lines 1–15). The FIC of each channel P_i is stored in variable $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$, which is computed by the inner loop (lines 3–10). Then, $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$ is processed based on whether we want to generate the full set of FIC or only SDFIC, which depend only on state variables (lines 11–14). The inner loop from lines 3 to 10 performs the main computation of FIC of channel P_i. Variable $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$, initialized to *true*, keeps a superset of P_i's FIC. Because FIC involve all state and output functions, the algorithm repeatedly refines $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$ on line 10 by taking the conjunction of $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$ and $\widetilde{FIC}_{P_i}(h)$, which is the FIC of P_i with respect to only one state transition (or output) function h. The innermost loop from lines 5 to 9 computes $\widetilde{FIC}_{P_i}(h)$ by deriving the set of unobservability conditions[5] with respect to h for each input variable $p_k^i \in P_i$ and intersecting all the sets across all the variables. After the loop terminates, line 10 applies the consensus function to eliminate any cube that contains input variables from channel P_i. These cubes may appear due to the conjunctions across the unobservability conditions of the single variables.

Lines 11 to 14 process $\widetilde{FIC}_{P_i}(\mathbf{f},\mathbf{g})$, depending on whether we require the complete set of FIC or SDFIC. In the former case, we augment the literals of input variables with conditions which are true if and only if the corresponding input channel presents valid data. Recall that valid 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 $qEmpty = 0$).

Alternatively, line 14 restricts FIC dependency to state variables only (SDFIC) by applying the consensus function over all input variables iteratively. SDFIC may be preferable because 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.

The final FIC of channel P_i is a single-output Boolean function. The domain of the complete $FIC_{P_i}(\mathbf{f},\mathbf{g})$ is the set of state variables, input variables, $voidIn$ and $qEmpty$ variables minus the set of inputs, $voidIn_i$ and $qEmpty_i$ variables of channel P_i. Instead, the domain of $SDFIC_{P_i}(\mathbf{f},\mathbf{g})$ includes only the state variables. Either function can be implemented as a combinational logic network like the channel FIC-detect block: At each clock cycle, $(FIC_{P_i}(\mathbf{f},\mathbf{g}) = 1)$ (or $SDFIC_{P_i}(\mathbf{f},\mathbf{g}) = 1$) if and only if the current data value of channel P_i is not needed to compute the state transition and the output function of the core.

Complexity: The time complexity of the algorithm depends on the data structure used to represent the Boolean functions and the method used to compute the unobservability conditions on line 7. We used the algorithm proposed in [16], where Boolean functions are represented as BDD [17]. Each iteration of the innermost loop (lines 5 to 9) takes $O(|E||G|^2) + O(|G|^2) = O(|E||G|^2)$ time, where $|G|$ is the size of the

[5]Computing unobservability conditions is the basis of our procedure, but not the focus of this paper. We refer the interested reader to [10].

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

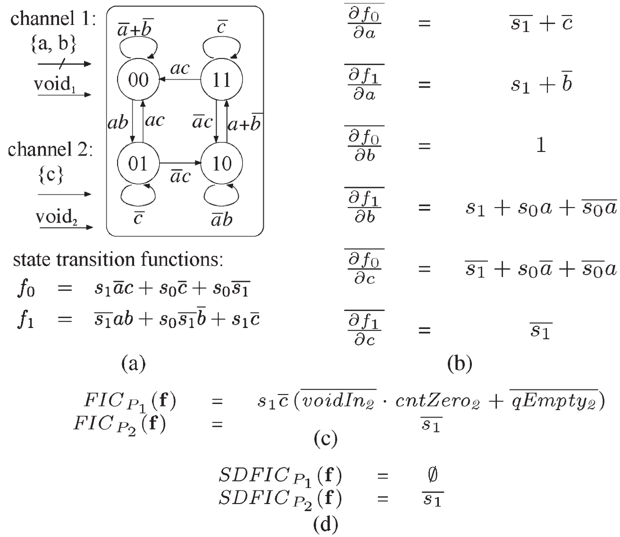


Fig. 10. (a) A core with two input channels (three input variables in total) modeled by a four-state Moore FSM. (b) Unobservability conditions of each input variable with respect to the two-state transition functions. (c) FIC depending on both inputs and states. (d) FIC depending on states only (SDFIC).

largest BDD and E is the set of edges of the Boolean network implementing the core. The innermost loop's total run time is $O(|P_i||E||G|^2)$. Since line 10 takes $O(|P_i||G|^2)$, the run time from lines 3 to 10 is $O((|FF| + |PO|)|P_i||E||G|^2)$, where FF/PO denote the sets of flip-flops/primary outputs of the core, respectively. Lines 11 to 14 take $O(|P_i||G|^2)$ in either case. In summary, the overall run time of our algorithm is $O(\sum_i(|FF| + |PO|)|P_i||E||G|^2) = O(|PI|(|FF| + |PO|)|E||G|^2)$, where PI is the set of primary inputs. Although, in the worst case, $|G|$ can be exponential in terms of $|PI| + |FF|$, in practice, its size is often manageable, as confirmed in our experiments. Also, compared with the algorithm given in [18], the new algorithm of Fig. 9 generates much smaller $|G|$ and thus runs faster. If we bound the size of $|G|$, the algorithm is polynomial to the size of the core.

Example: We apply the procedures discussed earlier to a simple core module whose behavior is modeled by a Moore FSM. The core, its FSM model, and the state transition functions are shown in Fig. 10(a). It has two input channels consisting of three variables in total ($\{a, b\}$ and c) and four states encoded as $(s_0 s_1) \in \{00, 01, 10, 11\}$. We applied our algorithm to derive the FIC for each input channel. The unobservability conditions of all three input variables with respect to each state transition function are shown in Fig. 10(b). After line 12, the FIC for each of the two channels are as follows: $FIC_{P_1}(\mathbf{f}) = s_1\overline{c}(\overline{voidIn_2} \cdot cntZero_2 + \overline{qEmpty_2})$ and $FIC_{P_2}(\mathbf{f}) = \overline{s_1}$. If we prefer to restrict ourselves to SDFIC, then we apply line 14. In this case, the FIC for channel 2 becomes $SDFIC_{P_2}(\mathbf{f}) = \overline{s_1}$, while the input data coming at channel 1 are always needed: $SDFIC_{P_1}(\mathbf{f}) = \emptyset$. Overall, less opportunities for avoiding stalling can be exploited, but this might help 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 [19]. We test it with a suite of sequential circuits including the ISCAS-89 benchmarks, and with a real-world SoC, an ultrawideband baseband transmitter [20], [21]. 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 assumption will be later discarded when we apply FIC optimization to the SoC). As defined in Section II, we distinguish a FIC that depends only on the core's state variables (SDFIC) from one that depends also on input variables (ISDFIC). The distributions in Fig. 11 illustrate the occurrence frequencies of FIC in reachable states for benchmark circuit *s1488*, specifically the ratio of reachable states in which a particular input has FIC [Fig. 11(a)], the number of inputs which have FIC in each of the 48 reachable states [Fig. 11(b)], and the ratio of states where at least some inputs have SDFIC [Fig. 11(c)]. The analysis only considers *satisfied* SDFIC at each reachable state for a given input. In circuit *s1488*, all but two inputs have satisfied SDFIC in most states. Furthermore, in most reachable states, there are a significant number of inputs which have FIC. The conclusions are that SDFIC are very frequent, and by considering also ISDFIC, only a little more FIC can be exploited (as indicated by the upper portion of each bar).

Fig. 12 shows the occurrence frequencies of FIC across all benchmarks. For each benchmark, columns "PI," "PO," and "FF" report the numbers of primary inputs, primary outputs, and flip-flops, respectively; column "# of inputs with SDFIC" reports the number of inputs which have satisfied SDFIC in at least one reachable state, while column "states with SDFIC inputs" reports the number of reachable states in which at least one input has one satisfied SDFIC. The nonweighted average of inputs with satisfied SDFIC per reachable state is given in the following column. The analogous analysis is applied to FIC (by considering both SDFIC and ISDFIC), 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 SDFIC is a subset of FIC, the number of SDFIC is high in most designs (e.g., all of FIC discovered in circuit *s349* are SDFIC). These results confirm that, *in practice, it is sufficient to focus on exploiting SDFIC* since they already offer many opportunities to improve the performance of a latency-insensitive system. Furthermore, the SDFIC-detect logic is typically faster and much smaller.

B. Case Study: An SoC for Wireless Communication

In the second set of experiments, we applied LID and the proposed FIC optimization to the semi-custom design of an SoC for wireless communication to measure the performance improvements made possible by FIC and assess the associated

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

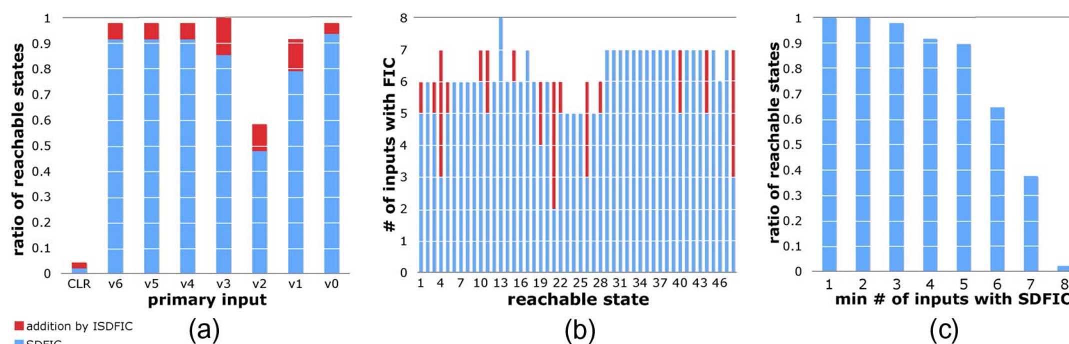


Fig. 11. Frequency distributions of FIC in s1488. In Figs. 11 and 12, the acronym "ISDFIC" refers to FIC that are functions of both state variables and at least one input variable, while "SDFIC" refers to FIC that are functions of state variables only.

bench	PI	PO	FF	reachable states	# of inputs with SDFIC	states with SDFIC inputs (%)	avg. # of inputs with SDFIC per state	# of inputs with FIC	states with FIC inputs (%)	avg. # of inputs with 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 FIC across all benchmarks.

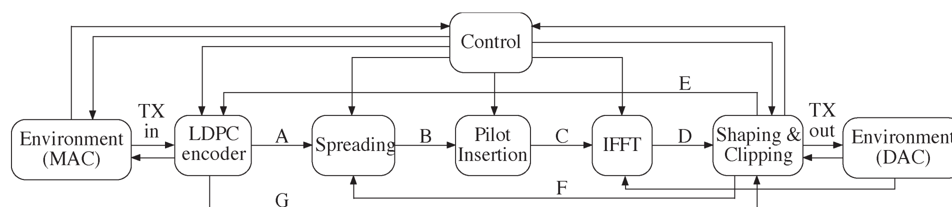


Fig. 13. LDPC-COFDM-based ultrawideband transmitter. The channels of the datapath are labeled alphabetically.

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 [20] and [21]: This is a "coded orthogonal frequency division modulation" (COFDM) baseband solution for ultrawideband systems. Fig. 13 shows the top-level diagram of the system: The transmitter receives packets from the medium access control 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; 2) an LID version of it; and 3) an 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[6] by exploiting the SDFIC 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 proposed FIC-detect synthesis algorithm finds these FIC in less than one second on a 3-GHz Pentium 4 machine with 1-GB memory. 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

[6]Modules with no SDFIC are encapsulated with classic shells. This is possible because the FIC-shell follows the same LI protocol as classic shells.

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

Fig. 14. Throughput speedup due to FIC-based optimization with one or two relay-station insertions on different channels.

40 000 clock cycles. To measure the area and delay, we did the following: 1) synthesized the three designs using Synopsys Design Compiler; 2) completed technology mapping with a 90-nm industrial standard cell library; and 3) performed static timing analysis on the mapped design.

Fig. 14 shows 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 very significantly in some cases, e.g., when one or two relay stations are inserted on channel F, the FIC optimization brings the throughput almost up to one, the ideal value. Overall, the throughput speedups across all configurations range from 0.3% to 30.7%; the average is 10.3%.

1) Effectiveness of FIC: All of the FIC are computed automatically without human interventions, and all but one module has at least one input channel with FIC (more precisely, SDFIC). Some of the FIC discovered by the algorithm are very effective. For example, the feedback channel F from the *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 an FIC roughly depends on how often it can be used to avoid stalling of modules in the critical loops. Fig. 15 shows 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 channels 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 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. In general, inserting more relay stations on a channel decreases the throughput of a latency-insensitive system, regardless of the use of FIC optimization. Indeed, since the FIC occurrence frequencies are independent from the number of relay stations, more relay stations induce more stalls that eventually cannot be mitigated by FIC.

2) 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 one, and 3.26% for shells with queue size of two. FIC-shells with FIC-detect blocks add negligible area (less than 0.01%) to the classical shells. The critical-path delays 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].

3) Sizing of Shift Registers: The charts of Fig. 16 report the system throughput of the COFDM design as a function of the size of the shift registers, which is varied uniformly from zero to three for all FIC-shells in the system. In particular, each curve on the left-hand side figure corresponds to the case when one relay station is inserted on one of the seven global data channels of Fig. 13. The curves on the right-hand side figure correspond to the case when two relay stations are inserted on the same channels. Note that the leftmost data points in the two charts correspond to the throughput value using classic LID shells because a FIC-shell with shift registers of size zero “degenerates” to a classic shell. The throughput improvement levels off when the size of the shift registers becomes larger than one in the case of one relay-station insertion and two in the case of two relay-station insertions.

4) FIC Optimization and Queue Sizing: As discussed in [8], [14], and [22], the input queue sizes in the shells also affect the system throughput. For any pair of reconvergent paths, if the number of relay stations on one path and the total queue capacity of the other are unbalanced, such reconvergent paths can become a critical *loop* consisting of forward datapaths and backward backpressure paths. This is because the cores on one of the reconvergent paths will periodically stall their uplink senders once their queues are full. For example, if we insert one relay station on channel F in an LID implementation of our design where queues have size one, the reconvergent path E–A–F becomes a critical loop with a throughput of $3/4 = 0.75$. To improve the throughput, one option is to increase the size of the shell’s input queues. For instance, increasing the queue sizes to two raises the throughput to 0.8. Columns labeled as “No FIC” in Fig. 17 report analogous throughput variations due to different queue sizes when the relay station is inserted on one of the channels.

On the other hand, the use of FIC creates more opportunities for throughput optimization. When a core utilizes the FIC of one of its input channels to avoid unnecessary stalls, the usage

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

RS	throughput		speedup	A's SDFIC		B's SDFIC		D's SDFIC		E's SDFIC		F's SDFIC	
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. 15. Throughput speedup due to FIC-based optimization with one relay-station insertion and shell queues of size two.

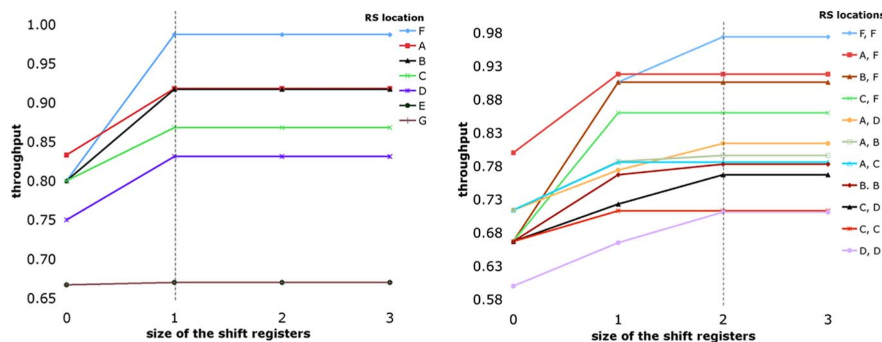


Fig. 16. Impact of the size of the shift registers in the FIC-shells on system throughput for the LID of the COFDM. (Left) One relay-station insertion. (Right) Two relay-station insertions.

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. 17. Impact of the FIC optimization and queue sizing on the throughput (with one relay-station insertion).

of FIC also reduces the chance of filling up the queues of the core's remaining input channels. Therefore, FIC can avoid stalls caused not only by the absence of valid data but also by backpressure. For instance, a FIC of channel F is enough to bring the throughput back to 0.98. This achieves higher throughput than the queue-sizing technique. In other scenarios, e.g., if the relay station is inserted on channel B, combining 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 Fig. 17 report the throughput data for the various scenarios.

5) Evaluations of FIC-Queue: We compared the FIC optimization using the FIC-queue technique to the one without using it, whose results are presented earlier. We found that the 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 channels A and E. The storage space of queues in each shell is one. The throughput of the design increases from 0.60 to 0.66. If we apply the FIC-queue technique to other design configurations, the throughput remains the same.

C. Discussion of FIC-Based Optimization

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. Still, the presence of FIC is mostly due to the *behavior* of a design, not to the suboptimality of the implementation of its logic circuits. That is, the behavior of a core module or the entire system *implicitly* introduces the FIC. 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 Fig. 11, is an add–shift multiplier [23] 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 [24]: 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 shown in Fig. 12, measuring the occurrence frequencies of FIC for the same suite of benchmarks after applying state minimization with STAMINA [25] and the logic optimization scripts in ABC.[7] The results are shown in Fig. 18.[8] Comparing this set of results with those of Fig. 12 shows that the FIC occurrence frequencies are almost the same. This

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

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

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

bench	PI	PO	FF	reachable states	# of inputs with SDFIC	states with SDFIC inputs (%)	avg. # of inputs with SDFIC per state	# of inputs with FIC	states with FIC inputs (%)	avg. # of inputs with 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 FIC across all benchmarks subjected to state minimization and sequential and combinational logic optimizations. Benchmarks whose state space cannot be handled by the tool are marked by dashes in the corresponding rows.

means that logic optimization does not significantly affect the number of FIC. Similarly, while the synthesis of the COFDM design that is returned by Synopsys Design Compiler includes various logic optimization steps, our procedure identifies FIC that are induced by the COFDM communication protocol. This is not 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 SoC automatically, without the knowledge of its logic 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 [26] and [27]. Early evaluation has been applied to phased logic at different granularity levels by Reese *et al.* [28], [29] and to the optimization of pipelined asynchronous logic by both Brej and Garside [30] and, more recently, by Ampalam and Singh [31].

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 multiclock latency-insensitive circuits in [32] and [33], and it has been applied to elastic systems by using a new latency-insensitive protocol that explicitly encodes antitoken signals [34]. The work by Casu and Macchiarulo on *adaptive* latency-insensitive protocols [13] and our preliminary results on FIC-based optimization [18] 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 as follows: 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).

1) Detection logic: To improve performance with early evaluation or exploiting FIC, a mechanism to dynamically detect the occurrence of such an event must be supplied. Most approaches in the literature assume that this mechanism is manually designed. The burden of manual design is partially reduced in the method described in [33], which, however, requires designers to provide high-level specifications of triggering functions that are then automatically translated into FSM implementations.

Reese *et al.* [28] 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.

Casu and Macchiarulo [13] 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. 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

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

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.

2) 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 that 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.* [28], [29] use Petri nets to model and implement such a handshaking mechanism for asynchronous phased-logic systems.

An alternative approach is to augment the communication infrastructure with flow of *antitokens*, which run in parallel with the normal data flow but in the opposite direction and annihilate unused (and unneeded) normal data tokens [31], [34]. Whenever a computation core early evaluates, it generates one antitoken for each input channel from which a late token is expected. Such mechanisms require communication protocols that accommodate the flow of antitokens as well as carefully designed interface circuits which propagate and destroy normal tokens and antitokens 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.* [35] for performance analysis of early evaluation. 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.[9]

Compared to the antitoken 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, and the antitoken techniques achieve the same effect by sending out antitokens continuously as long as there is no traffic congestion of the antitoken flows. Interestingly, compared to using antitokens, the counting-based method can be viewed as storing (the number of) the antitokens locally in queues, which provide buffering mechanism. Finally, while the communication interfaces supporting antitoken flows require the modification of the global communication protocols with the insertion of additional control signals, the counting-based methods do not as they only demand changes that are inherently "local" to the interface.

[9] Casu and Macchiarulo have proposed a novel technique to use FIC to reduce not only 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. Conclusion

We studied the problem of leveraging the local knowledge on the internal logic of a core to improve the global SoC performance in LID. We defined the notion of FIC, and we described a logic synthesis procedure to generate automatically a shell interface (a FIC-shell) around a given 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 semicustom 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%.

Acknowledgment

The authors would like to thank H.-Y. Liu and C.-Y. Lee for providing the RTL design of the COFDM SoC and C. Pinello for the 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*. Norwell, MA: Kluwer, 1999.

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

[4] L. Scheffer, "Methodologies and tools for pipelined on-chip interconnect," in *Proc. Int. Conf. Comput. Des.*, 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[6] 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.

[7] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proc. Des. Autom. Conf.*, Jun. 2000, pp. 361–367.

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

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.

[9] 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 Des.*, San Jose, CA, Nov. 1999, pp. 309–315.

[10] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 7, no. 6, pp. 723–740, Jun. 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 *Proc. Int. Conf. Formal Methods Models Codesign*, 2007, pp. 13–22.

[13] M. R. Casu and L. Macchiarulo, "Adaptive latency-insensitive protocols," *IEEE Des. Test Comput.*, vol. 24, no. 5, pp. 442–452, Sep./Oct. 2007.

[14] R. Collins and L. P. Carloni, "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system," in *Proc. Des. Autom. Conf.*, Jun. 2007, pp. 410–416.

[15] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 5, pp. 440–449, Sep. 1980.

[16] M. Damiani and G. D. Micheli, "Don't care set specifications in combinational and synchronous logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 3, pp. 365–388, Mar. 1993.

[17] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[18] C.-H. Li and L. P. Carloni, "Using functional independence conditions to optimize the performance of latency-insensitive systems," in *Proc. Int. Conf. Comput.-Aided Des.*, 2007, pp. 32–39.

[19] *ABC: A system for sequential synthesis and verification*. [Online]. Available: http://www.eecs.berkeley.edu/alanmi/abc/

[20] 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 480 mb/s LDPC-COFDM-based UWB baseband transceiver," in *Proc. ISSCC Dig. Tech. Papers*, 2005, vol. 1, pp. 444–609.

[21] C.-Y. Lee, H.-Y. Liu, and C.-C. Lin, "SoC for COFDM wireless communications: Challenges and opportunities," in *Proc. Int. Symp. VLSI Des., Autom. Test*, 2006, pp. 1–4.

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

[23] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Des. Test Comput.*, vol. 16, no. 3, pp. 72–80, Jul. 1999.

[24] 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.

[25] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 2, pp. 167–177, Feb. 1994.

[26] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny, "On the models for asynchronous circuit behaviour with OR causality," *Form. Methods Syst. Des.*, vol. 9, no. 3, pp. 189–233, Nov. 1996.

[27] A. Bystrov, D. Sokolov, and A. Yakovlev, "Low-latency control structures with slack," in *Proc. Int. Symp. Asynchr. Circuits Syst.*, May 2003, pp. 164–173.

[28] R. R. Reese, M. A. Thornton, and C. Traver, "A coarse-grain phased logic CPU," in *Proc. Int. Symp. Asynchr. Circuits Syst.*, 2003, pp. 2–13.

[29] R. R. Reese, M. A. Thornton, C. Traver, and D. Hemmendinger, "Early evaluation for performance enhancement in phased logic," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 4, pp. 532–550, Apr. 2005.

[30] C. F. Brej and J. D. Garside, "Early output logic using anti-tokens," in *Proc. Int. Workshop Logic Synth.*, 2003, pp. 302–309.

[31] M. Ampalam and M. Singh, "Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens," in *Proc. Int. Conf. Comput.-Aided Des.*, 2006, pp. 611–618.

[32] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proc. Conf. Des., Autom. Test Eur.*, 2004, pp. 1008–1013.

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

[34] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *Proc. Des. Autom. Conf.*, 2007, pp. 416–419.

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

Cheng-Hong Li received the B.S. degree in electrical engineering and the M.S. degree in computer science from National Taiwan University, Taipei, Taiwan, in 1998 and 2001 respectively. He is currently working toward the Ph.D. degree in the Department of Computer Science, Columbia University, New York, NY.

His research interests include communication-based design methodology of systems-on-chip, formal verification, and code compression for embedded systems.

Luca P. Carloni (S'95–M'04) 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, in 1997 and 2004, respectively.

He is currently an Assistant Professor with the Department of Computer Science, Columbia University, New York, NY. He has authored over 50 publications and is the holder of one patent. His research interests include design tools and methodologies for integrated circuits and systems, distributed embedded systems design, and design of high-performance computer systems.

Dr. Carloni is a member of the IEEE Computer Society. He received the Faculty Early Career Development (CAREER) Award from the National Science Foundation in 2006 and was selected as an Alfred P. Sloan Research Fellow in 2008. He is the recipient of the 2002 Demetri Angelakos Memorial Achievement Award "in recognition of altruistic attitude towards fellow graduate students." In 2002, one of his papers was selected for "The Best of ICCAD," a collection of the best IEEE International Conference on Computer-Aided Design papers of the past 20 years.

Authorized licensed use limited to: Columbia University. Downloaded on February 18,2010 at 20:35:46 EST from IEEE Xplore. Restrictions apply.