Using Functional Independence Conditions to Optimize the Performance of Latency-Insensitive Systems

Cheng–Hong Li and Luca P. Carloni
Department of Computer Science - Columbia University in the City of New York

***Abstract*—In latency-insensitive design *shell* modules are used to encapsulate system components (*pearls*) in order to interface them with the given latency-insensitive protocol and dynamically control their operations. In particular, a shell stalls a pearl whenever new valid data are not available on its input channels. We study how functional independence conditions (FIC) can be applied to the performance optimization of a latency-insensitive system by avoiding unnecessary stalling of their pearls. We present a novel circuit design of a generic shell template that can exploit FICs. We describe an automatic procedure for the logic synthesis of a FIC-shell instance that is only based on the analysis of the logic structure of its corresponding pearl and does not require any input from the designers. We implemented the proposed technique within the logic synthesis tool ABC and we use it to complete various experiments that demonstrate its performance benefits and limited overhead. In particular, we completed the semi-custom design of a system-on-chip (SoC), an ultra-wideband baseband transmitter, using a state-of-the-art $90nm$ technology process. To the best of our knowledge this represents the first report on the complete latency-insensitive design of a real-world SoC.**

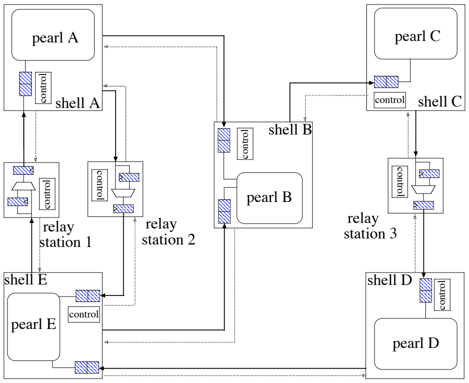


Fig. 1. Shell encapsulation, relay station insertion, and channel back-pressure.

I. Introduction

Latency-insensitive design (LID) is a correct-by-construction approach that handles latency's increasing impact on nanometer technologies and facilitates the reuse of intellectual-property cores for building complex systems-on-chip (SoC), thereby reducing the number of costly iterations in the design process to achieve timing closure [7], [9]. In particular, it provides a sound way to address the problem of interconnect delay in nanometer design by simplifying the application of wire pipelining in the context of traditional design practice that are based on the *synchronous paradigm*. A functionally-equivalent latency-insensitive system can be derived from an original synchronous one by encapsulating any sequential logic block (*pearl* or *core*) within an automatically generated interface process (*shell*). While the pearl can be an arbitrarily-complex sequential module (a finite state machine, a pipelined circuit,...), the only requirement is that it is *stallable*, i.e. it can be *clock gated*. Figure 1 shows a latency-insensitive system with five shell-pearl pairs connected by point-to-point, unidirectional channels. At the implementation stage, 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 capacity of at least two and simple flow control logic. The shell logic and relay stations together implement a latency-insensitive protocol [7] that is designed to accommodate arbitrary variations of wire delays while guaranteeing that the functional behavior of the original synchronous system is preserved (*semantics preservation*). Data communicated over a channel is labeled by a bit signal indicating whether the data is valid or void at a given clock cycle. At each cycle the shell fires the pearl if and only if each input channel presents a new valid data token (*AND-firing semantics*). Otherwise, it *stalls* the pearl through clock gating while putting void data on each output channel.

LID helps to meet the required target clock frequencies through automatic wire pipelining [10], [11], [14], [22] 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 [8], [20]. This is because each relay station must be initialized with a void data token (a "bubble" or τ). If the relay station is inserted on a cyclic path, such as a feedback loop, the bubble circulates in the loop indefinitely, thus causing the overall system throughput to drop below the ideal value (equal to one). For example, the two relay stations placed between Pearl A and E in Fig. 1 induce two bubbles circulating in the loop that stall these pearls 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 [8], [20].

In this paper, we study how *functional independence conditions (FIC)* of sequential pearls from the input variables can be applied to the performance optimization of a latency-insensitive system by avoiding unnecessary stalling of their pearls. Basically, whenever an input data value is not needed for the current computation of the pearl and *even if no valid data token is present on the corresponding channel* the pearl could still be fired. Thus the number of stalls incurred in the whole system could be reduced. Such FICs [1] 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.

A Simple Motivating Example. Consider the synchronous system of Fig. 3 having two interconnected Moore FSMs M_1 and M_2. Each FSM has one 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 present three states: the set of states of M_1 is $\{A, B, C\}$ and the 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

[1] We prefer to 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.

1-4244-1382-6/07/$25.00 ©2007 IEEE

		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}	$\dots$
Strict System	X:	a	c	a	a	b	a	c	a	a	b	a	c	a	a	b	a	$\dots$
	Y:	d	f	e	f	e	d	f	e	f	e	d	f	e	f	e	d	$\dots$
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	$\dots$
	Y_b':	$\boxed{\tau}$	d	f	$\boxed{\tau}$	e	f	$\boxed{\tau}$	e	d	$\boxed{\tau}$	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	$\dots$
	Y_b:	d	f	$\boxed{\tau}$	e	f	$\boxed{\tau}$	e	d	$\boxed{\tau}$	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	$\dots$
	stalling:	M_1	M_2	—	M_1	M_2	—	M_1	M_2	—	M_1	M_2	—	M_1	M_2	—	M_1	$\dots$
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	$\boxed{\tau}$	$\dots$
	Y_b':	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	$\dots$
	Y_b:	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	e	$\boxed{\tau}$	f	e	$\boxed{\tau}$	d	f	$\dots$
	stalling:	M_1	—	(M_2)	—	M_1	M_2	—	M_1	—	(M_2)	—	M_1	M_2	—	M_1	—	$\dots$

Fig. 2. Set of traces for the behaviors of the three systems in the motivating example.

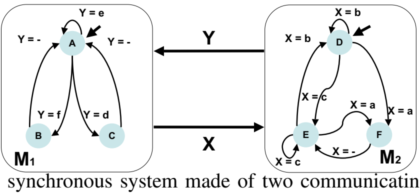


Fig. 3. A synchronous system made of two communicating FSMs.

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 Fig. 2: the first set describes the behavior of the strictly synchronous system of Fig. 3. It is easy to see that the system cycles according to a periodic sequence of five compound state transitions: for M_1 we have $(A \rightarrow C \rightarrow A \rightarrow A \rightarrow B) \rightarrow (A \rightarrow C \dots)$, while for M_2 we have $(D \rightarrow F \rightarrow E \rightarrow F \rightarrow E) \rightarrow (D \rightarrow F \dots)$.

The second set of traces in Fig. 2 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 the corresponding 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 [7]: i.e., they are the same if one ignores the τ symbols. But, the system throughput is reduced from 1 to $\frac{2}{3} = 0.66$.

Part of the lost throughput, however, can be recovered if one knows the internal structure of the FSM (an assumption not made in [7] where pearls are treated as *black boxes*). For instance, the transition of $M2$ from state F is functionally independent from the value of input X. 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 Fig. 2 where the stalled FSM is reported in the last row (and delayed stalls are marked with parenthesis). The key point is that, for this simple system, delaying one stall by only a single clock cycle allows us to raise the throughput by 9% to $\frac{5}{7} = 0.72$.

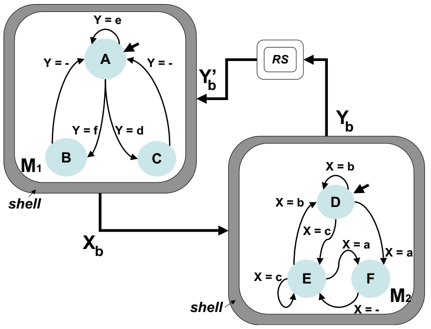


Fig. 4. A latency-insensitive system derived from the system of Fig. 3

Contributions. In the next pages we present a new circuit design of a generic shell template that can dynamically exploit FICs when the pearl is given as a *white box*. We also provide a *fully automatic* procedure for the logic synthesis of a shell instance based on the particular characteristics of its corresponding pearl. Our method requires no input from designers and relies on efficient logic synthesis algorithms. Finally, we present the first empirical study of the applicability and effectiveness of optimizations based on FICs in the LID methodology including a report on the semi-custom design of a real-world 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.

II. Related Work

In the asynchronous design community the concept of *early evaluation* has been proposed to allow a logic component to compute its output before all of its input values are available: Reese *et al.* apply "early evaluation" to phased logic in different granularities [23], [24] while Ampalam *et al.* [3] and Brej *et al.* [5] use "anti-tokens" to support early evaluations in pipelined asynchronous logic. Arguably, in this paper we apply early evaluation to the optimization of *synchronous* systems in the context of the latency-insensitive design methodology [6], [7]. Specifically we start from a synchronous specification such as a network of FSMs and automatically derive a synchronous latency-insensitive implementation. To start from a synchronous specification offers many practical advantages in the design of complex integrated circuits as explained in [6], [7]. A related method for the optimization of latency-insensitive systems in the presence of multi-clock domains is proposed in [2], [25].

To exploit functional independence, a detection logic triggering an early evaluation must be supplied. Reese *et al.* present an algorithm based on traversing root-to-terminal paths in BDDs that is suitable for synthesizing one trigger function on a fixed subset set of inputs [24]. We propose a scalable algorithm that uses observability don't-cares to target arbitrary multi-input and multi-output logic functions. This algorithm finds all the triggering conditions on all of the possible subsets of inputs.

One challenge of exploiting functional independence to allow early evaluations/outputs is to ensure a system's functional correctness. 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 (and thus it will potentially cause delayed stalling). Hence, it is necessary to ensure that all the computations are fired on the fresh data tokens. We achieve this goal by recording the number of subsequent tokens to be discarded for each input channel. This idea is similar to the notion of "negative tokens" in the "guarded" Petri net model [15]. To implement it we use simple and efficient hardware (a 1-bit shift register). In [23], [24], computation correctness is ensured by acknowledging early and late arrival data tokens simultaneously. In [3], [5] early evaluation generates *anti-token* flowing in the opposite direction of normal data flows to cancel unused (and unneeded) normal tokens. One drawback of the method proposed in [23], [24] is that a component which early-evaluates must still wait for the arrival of all the inputs before proceeding to its next computation. This restriction is lifted both in our approach and in [3], [5] where more frequent back-to-back early firings are possible. While in [3], [5] modified pipeline protocols must be adopted, our method is *local* to the pearl modules and does not change the global communication protocol.

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 [6], [17], which we review first.

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 pearl module is *latency-equivalent* to the original pearl module. Conceptually a shell has two different kinds of logic controllers (though in implementation they can be combined): a *firing control block* deciding when a pearl module should be stalled by gating the pearl's clock, and a *channel control block* that handles incoming data tokens, interface signals, and input queue operations for each channel. A shell receives data from input channels and broadcasts outputs of the pearl 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 ("back-pressure") [6], [17].

At each clock cycle the shell decides whether the computation of a pearl module can proceed: the computation is allowed for the next clock cycle ("firing") when all of the input channels are *ready* (in a classic shell an input channel is ready if it presents a valid data token). Otherwise the shell stalls the pearl by gating the clock with signal *fire_next*. The output tokens generated by a stalled module are marked as void. When a void data token is received, it is discarded. Valid tokens not consumed (due to stalling) are stored in queues for later use. Thus a valid data token can come either directly from the input channel or from the queue. In either case, the channel is declared *ready*. The shell also stalls the pearl whenever a downstream receiver notifies that it cannot consume more tokens by asserting the stop signal [6], [17].

FIC-shell design. Fig. 5(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 I. 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 FICs can be exploited, i.e. whenever one or more input channels present invalid data tokens which are not necessary to the pearl's computation. In this case, these channels are declared ready and the FIC-shell fires the pearl module. However, this operation makes the pearl run one more clock cycle *ahead* of the next valid data token for such channels. So, when this token arrives it must be discarded. Therefore, for each input channel a FIC-shell maintains a counter that records how many cycles the pearl module currently runs ahead with respect to the next valid data token on the channel. The detailed logic of the FIC-shell is reported in Fig. 5(b). In summary, a channel is declared ready if either it has a valid and fresh token (the channel count is zero), or it carries a value that is not necessary for the computation. The count is maintained by simple rules. Whenever a pearl is fired if a channel presents a void token its counter is incremented by 1. A non-zero count indicates the next valid data token is outdated, and it should be discarded on arrival (causing a delayed stall). When a valid token is dropped in this case, the count is decreased by 1. In practice, instead of using an up-down counter, a shift register is sufficient because the actual count is not needed. The count is increased by shifting a "1" into the register (*sh_right*), and decreased by shifting a "1" out (*sh_left*). The leftmost bit indicates whether the count is zero (*sr_empty*), and the rightmost bit flags whether the register reaches its maximum capacity (*sr_full*). When the shift register is full, each controller can no longer declare as ready its channel even if this channel is receiving a void token under a FIC condition.[2] So regardless of the size of the shift register, the FIC-shell will always be able to synchronize the incoming data tokens properly. Whether a FIC occurs on a given channel at a given clock cycle is dynamically established by the *FIC-detect* block: this is a combinational logic block that monitors the current state of the pearl and all the input channels. Each channel has its own single-output FIC-detect block.[3] When the FIC-detect evaluates to 1, the current data token of the channel is a FIC. In Section IV we present a procedure for the logic synthesis of this block.

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 pearl 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. Because the system throughput is dominated by the critical cycle(s) [8], [20], FIC-shells can be used only for those pearl modules that are part of feedback loops, while classic shells are sufficient elsewhere.

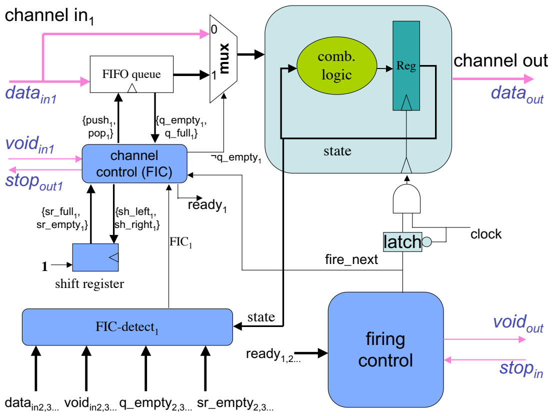
IV. Logic Synthesis of FIC-Detect Block

We present a procedure to automatically synthesize the logic of the channel FIC-detect blocks. For each input channel $\mathbf{P}_i$, which may consist of many binary variables, we first derive the conditions under which the state transition and output functions of the pearl module do not depend on $\mathbf{P}_i$. These functional independence conditions (FIC) are generally expressed as predicate on the set of the pearl's state variables together with the remaining input variables and their validity (decided by the *void* and *q_empty* signals). Our procedure uses observability don't cares as a starting point for FIC computation.[4]

[2] However, a channel is declared as ready if a valid token is received and the token is not needed (FIC) when the shift register is full. This is because the count maintained by the shift register is not increased in this case: this valid token is dropped regardless of whether the pearl will be fired, so the count will either be the same (if the pearl is fired) or will be decreased (not fired).

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

[4] ODC computation is the basis of our procedure, but not the focus of this paper. For ODC we refer the interested reader to [21].



(a)

$$\begin{array}{rcl}
\forall i \in \mathcal{I}\ \mathit{ready}_i & = & \overline{\mathit{void\_in}_i} \cdot \mathit{sr\_empty}_i + \overline{\mathit{q\_empty}_i} + \\
 & & \mathit{FIC}_i \cdot (\overline{\mathit{sr\_full}_i} + \overline{\mathit{void\_in}_i}) \\
\mathit{stop\_out}_i & = & \mathit{q\_full}_i \\
\mathit{push}_i & = & \mathit{sr\_empty}_i \cdot \overline{\mathit{void\_in}_i} \cdot (\overline{\mathit{fire\_next}} + \overline{\mathit{q\_empty}_i}) \cdot \overline{\mathit{q\_full}_i} \\
\mathit{pop}_i & = & \overline{\mathit{q\_empty}_i} \cdot \mathit{fire\_next} \\
\mathit{sh\_left}_i & = & \overline{\mathit{void\_in}_i} \cdot \overline{\mathit{fire\_next}} \cdot \overline{\mathit{sr\_empty}_i} \\
\mathit{sh\_right}_i & = & \mathit{fire\_next} \cdot \mathit{void\_in}_i \cdot \mathit{q\_empty}_i \cdot \overline{\mathit{sr\_full}_i} \\
 & & \\
\mathit{fire\_next} & = & \bigwedge_{i \in \mathcal{I}} (\mathit{ready}_i) \cdot \overline{\bigvee_{j \in \mathcal{O}} (\mathit{stop\_in}_j \cdot \overline{\mathit{void\_out}_j})} \\
\forall j \in \mathcal{O}\ \mathit{void\_out}_j^{+} & = & \begin{cases} 0 & \text{if } \mathit{stop\_in}_j \cdot \overline{\mathit{void\_out}_j} \text{ is true} \\ \overline{\mathit{fire\_next}} & \text{otherwise} \end{cases}
\end{array}$$

(b)

Fig. 5. (a) Block diagram of a FIC-shell. (b) FIC-detect logic for channel i.

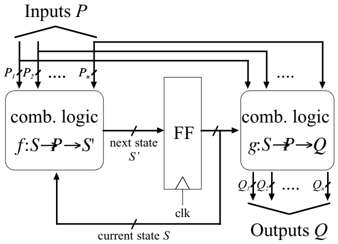


Fig. 6. Modeling of a pearl module.

We then use the FICs to synthesize the FIC-detect block for the channel. Before presenting our procedure we recall some background concepts.

Background Definitions. Without loss of generality a pearl module can be modeled as a Mealy FSM that is specified by a *state transition function* f, and an *output function* g (Fig. 6.). Note that a pipelined synchronous circuit also fits into this model by separating the combinational network and sequential elements. Further, a Moore FSM, where outputs depend only on states, can be viewed as a special case of a Mealy FSM. A generic state transition function can be written in vector form as:

$$\mathbf{S}' = \mathbf{f}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n; \mathbf{S})$$

where $\mathbf{S}' \equiv \{s'_1, s'_2, \ldots, s'_n\}$ is the vector of next state variables, $\mathbf{P}_i \equiv \{p_{i1}, p_{i2}, \ldots, p_{i_{|\mathbf{P}_i|}}\}$ is an input channel consisting of variables $p_{i_1} \ldots, p_{i_{|\mathbf{P}_i|}}$, and $\mathbf{S}$ is the vector of the present-state variables. The FSM output function is specified as

$$\mathbf{Q} = \mathbf{g}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n; \mathbf{S})$$

For a Boolean function f, a variable x_i is an *observability don't care (ODC)* if f is not sensitive to the changes of x_i [21]. ODCs may only hold under certain conditions that are expressed by the complement of the *Boolean difference*, which computes under which conditions f is sensitive to x_i. The Boolean difference is simply the result of *XOR* ($\oplus$) of f's co-factor with respect to x_i and $\overline{x_i}$. Let $\mathrm{ODC}_{x_i}(f)$ be the conditions under which function f is insensitive to variable x_i. We have

$$\mathrm{ODC}_{x_i}(f) = \overline{\frac{\partial f}{\partial x_i}} = f|_{x_i=1} \,\overline{\oplus}\, f|_{x_i=0}$$

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

Computing ODC using Boolean difference directly on a large multi-level Boolean network may not be practical, unless the network's global logic function $\mathbf{f}$ (which maps primary inputs directly to outputs) is given, or can be efficiently derived. An effective solution, which has been shown successful on large designs, is to iteratively applying Boolean difference functions locally [21]. For simplicity, in the sequel the Boolean difference will still be used as a notation to represent the computation of ODC sets.

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

$$C_{x_i}(f) = f|_{x_i=1} \cdot f|_{x_i=0} \tag{1}$$

Consensus can be extended to a set of variables by iteratively applying Eq. 1 to each variable [21].

Synthesis Procedure of FIC-Detect Block. The procedure consists of four steps:

Step 1. To derive the FICs for an input channel $\mathbf{P}_i$, we first restrict the computation to a single input variable $p_{i_j} \in \mathbf{P}_i$ with respect to a scalar state transition function $f_{s'_k}$ ($s'_k \in \mathbf{S}'$ is a single next state variable). We have:

$$\widetilde{\mathrm{ODC}}_{p_{i_j}}(f_{s'_k}) = \overline{\frac{\partial f_{s'_k}}{\partial p_{i_j}}} = f_{s'_k}|_{p_{i_j}=1} \,\overline{\oplus}\, f_{s'_k}|_{p_{i_j}=0} \tag{2}$$

Similarly for the FICs of p_{i_j} w.r.t. output function g_{q_l} we have:

$$\widetilde{\mathrm{ODC}}_{p_{i_j}}(g_{q_l}) = \overline{\frac{\partial g_{q_l}}{\partial p_{i_j}}} = g_{q_l}|_{p_{i_j}=1} \,\overline{\oplus}\, g_{q_l}|_{p_{i_j}=0} \tag{3}$$

Step 2. Since FICs involve all state and output variables we perform the conjunction of all the FICs computed by Eq. (2) and Eq. (3):

$$\widetilde{\mathrm{ODC}}_{p_{i_j}}(\mathbf{f}, \mathbf{g}) = \Big(\bigwedge_{s'_k \in \mathbf{S}'} \mathrm{ODC}_{p_{i_j}}(f_{s'_k})\Big) \cdot \Big(\bigwedge_{q_l \in \mathbf{Q}} \mathrm{ODC}_{p_{i_j}}(g_{q_l})\Big) \tag{4}$$

Step 3. A channel $\mathbf{P}_i$ has generally many input variables. Hence, we take the conjunction across all of them to determine its exact FICs:

$$\begin{aligned}
\widetilde{\mathrm{ODC}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g}) &= C_{\mathbf{P}_i}\Big(\bigwedge_{p_{i_j} \in \mathbf{P}_i} \mathrm{ODC}_{p_{i_j}}(\mathbf{f}, \mathbf{g})\Big) \\
&= C_{p_{i_1}}(C_{p_{i_2}}(\cdots C_{p_{i_j}}(\bigwedge_{p_{i_j} \in \mathbf{P}_i} \mathrm{ODC}_{p_{i_j}}(\mathbf{f}, \mathbf{g}))\cdots)
\end{aligned} \tag{5}$$

Note that the consensus function is used to eliminate any cube that contains input variables from channel $\mathbf{P}_i$. These cubes can arise after taking the conjunction of the single-variables FICs.

Step 4. In LID not every input channel presents a good token at each clock cycle. So we require all the input variables which appear in Eq. (5) to come from input channels presenting valid tokens. Recall that a good token can come either from the channel (i.e. its *void* is 0) or from the channel's queue (i.e. the queue is not empty). Further, if the token is from the channel, it cannot be outdated (shift register must be empty). So the final FICs can be obtained as follows:

$$\begin{aligned}\mathrm{ODC}^{\mathtt{IS}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g}) \equiv{}& \text{Replace each literal } p \text{ in } \widetilde{\mathrm{ODC}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g}) \text{ with} \\ & p\cdot(\overline{void_k}\cdot sr\_empty_k+\overline{q\_empty_k}), \text{ and } \overline{p} \\ & \text{with } \overline{p}\cdot(\overline{void_k}\cdot sr\_empty_k+\overline{q\_empty_k})\end{aligned} \qquad (6)$$

where $void_k$ and q_empty_k are the void and queue's empty signals of channel $\mathbf{P}_k$ containing variable p, while sr_empty_k is the shift register empty signal.

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

FICs that depend on input channels may induce extra timing constraints. In fact, the firing of a pearl module is controlled by the *fire_next* signal, which must be stable by the end of each clock cycle. The dependency of FICs on input and void variables may lead to long combinational paths from the sender of data tokens to *fire_next* across the communication channel. Therefore, we may want to restrict ourselves to FICs depending only on state variables. This requires a different (alternative) final step in our procedure.

Step 4'. To restrict FICs to state variables only, we apply the consensus function to Eq. (5) over all input variables iteratively:

$$\begin{aligned}\mathrm{ODC}^{\mathtt{S}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g}) &= C_{\mathbf{P}}(\widetilde{\mathrm{ODC}}_{p_i}(\mathbf{f},\mathbf{g})) \\ &= C_{p_{1_1}}(C_{p_{1_2}}(\cdots C_{p_{i_j}}(\widetilde{\mathrm{ODC}}_{p_i}(\mathbf{f},\mathbf{g})\cdots)))\end{aligned} \qquad (7)$$

If the pearl module has no combinational path from its inputs to outputs (thus it can be viewed as a Moore FSM), Eq. (3) will return 1 because an output variable does not depend on any input. The same steps can still be applied thereby $\mathrm{ODC}^{\mathtt{S}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g})$ is simply $\mathrm{ODC}^{\mathtt{S}}_{\mathbf{P}_i}(\mathbf{f})$.

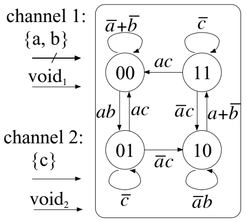
Example. The procedures discussed above is applied to a simple pearl module whose behavior is modeled by a Moore FSM. The pearl, its FSM model, and the state transition functions are reported in Fig. 7(a). The pearl has two input channels consisting of three variables in total ($\{a, b\}$ and c), and the FSM has four states ($s_0 s_1 = \{00, 01, 10, 11\}$).

We applied our four-step procedures to derive the FICs for each input channel. Since the pearl is a Moore FSM, only Eq. (2) must be applied in Step 1. The FICs of all three input variables with respect to each state transition function are shown in Fig. 7(b). Finally, Eq. (4) and Eq. (5) provide the FICs for each of the two channels: $\mathrm{ODC}^{\mathtt{IS}}_{\mathbf{P}_1}(\mathbf{f}) = s_1\overline{c}\,(\overline{void_2}\cdot sr\_empty_2+\overline{q\_empty_2})$ and $\mathrm{ODC}^{\mathtt{IS}}_{\mathbf{P}_2}(\mathbf{f}) = \overline{s_1}$.

If we prefer to restrict ourselves to FICs depending only on the state variables, then we apply Step 4' instead of Step 4. In this case, the FIC for channel 2 becomes $\mathrm{ODC}^{\mathtt{S}}_{\mathbf{P}_2}(\mathbf{f}) = \overline{s_1}$, while the input data coming at channel 1 are always needed: $\mathrm{ODC}^{\mathtt{S}}_{\mathbf{P}_1}(\mathbf{f}) = \emptyset$. 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 [1]. 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 [16], [18]. Both experiments demonstrate that FIC-based optimization has broad applicability, is efficient, and imposes little overhead.



state transition functions:

$$\begin{aligned} f_0 &= s_1\overline{a}c + s_0\overline{c} + s_0\overline{s_1} \\ f_1 &= \overline{s_1}ab + s_0\overline{s_1}\overline{b} + s_1\overline{c}\end{aligned}$$

(a)

$$\begin{array}{|lcc|} \hline \widetilde{\mathrm{ODC}}_a(f_0) & = & \overline{s_1}+\overline{c} \\ \widetilde{\mathrm{ODC}}_a(f_1) & = & s_1+\overline{b} \\ \widetilde{\mathrm{ODC}}_b(f_0) & = & 1 \\ \widetilde{\mathrm{ODC}}_b(f_1) & = & s_1+s_0a+\overline{s_0a} \\ \widetilde{\mathrm{ODC}}_c(f_0) & = & \overline{s_1}+s_0\overline{a}+\overline{s_0}a \\ \widetilde{\mathrm{ODC}}_c(f_1) & = & \overline{s_1} \\ \hline \end{array}$$

(b)

$$\begin{array}{|lcc|} \hline \mathrm{ODC}^{\mathtt{IS}}_{P_1}(\vec{f}) & = & s_1\overline{c}\,(\overline{void_2}\cdot sr\_empty_2+\overline{q\_empty_2}) \\ \mathrm{ODC}^{\mathtt{IS}}_{P_2}(\vec{f}) & = & \overline{s_1} \\ \hline \end{array}$$

(c)

$$\begin{array}{|lcc|} \hline \mathrm{ODC}^{\mathtt{S}}_{P_1}(\vec{f}) & = & \emptyset \\ \mathrm{ODC}^{\mathtt{S}}_{P_2}(\vec{f}) & = & \overline{s_1} \\ \hline \end{array}$$

(d)

Fig. 7. (a) A pearl module with 2 input channels (3 input variables in total) modeled by a 4-state Moore FSM. (b) The ODC sets of each input variable with respect to the 2-state transition functions. (c) The final FICs depending both on inputs and states. (d) The final FICs depending only on states.

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 functional independence conditions (FIC) are derived assuming that each single input is a LID channel (this overly-simplified assumption will be discarded when we apply FIC optimization to the real SoC later). We distinguish a FIC that depends only on pearl's state variables (SD-FIC) from one that depends also on input variables (ISD-FIC). Fig. 8 reports three distributions showing the occurrence frequencies of FICs in reachable states for benchmark circuit $s1488$. Fig. 8(a) lists the ratio of reachable states in which a particular input is a FIC. Fig. 8(b) lists the number of FIC inputs in each of the 48 reachable states. Fig. 8(c) shows the ratio of states where at least some number of inputs are SD-FIC. In benchmark circuit $s1488$, SD-FIC are very frequent: all but two inputs are SD-FIC in most states. Further, in most reachable states there is a significant number of inputs which are FICs. Note that SD-FICs dominate, and by considering also ISD-FICs only a little more functional independence conditions can be exploited.

Fig. 9 shows the occurrence frequencies of FICs across all benchmarks. For each benchmark, column "PI", "PO", "FF" report the number of primary inputs, primary outputs, and flip-flops respectively; column "# of SD-FIC inputs" reports the number of inputs which are SD-FIC in at least one reachable state, while column "states with SD-FIC" reports the number of reachable states in which at least one input is a SD-FIC. The non-weighted average of SD-FIC inputs per reachable states is given in the following column. The same analysis is applied to ISD-FICs, and results are listed in the last three columns. *These experimental results indicate that FICs 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 FIC inputs are SD-FIC in benchmark circuit $s349$. This is an add-shift-multiplier [13], controlled by a 3-bit counter. Its inputs are only needed in the first cycle of each computation round (thus state-dependent).

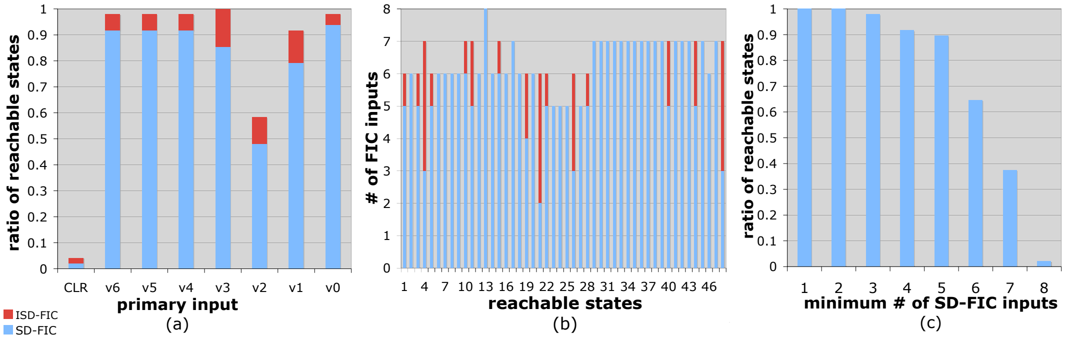


Fig. 8. Frequency distributions of functional independence conditions in s1488. In Fig. 8 and 9, the acronym "ISD-FIC" refers to FICs that are functions of both state variables and at least one input variable while "SD-FIC" refers to FICs that are functions of state variables only.

Bench	PI	PO	FF	reachable states	# of SD-FIC inputs	states with SD-FIC inputs (%)	avg. SD-FIC inputs per state	# of ISD-FIC inputs	states with ISD-FIC inputs (%)	avg. ISD-FIC inputs 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. 9. Statistics on the occurrence frequencies of functional independence conditions across all benchmarks.

These results confirm that in practice it is sufficient to focus on exploiting SD-FICs 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.

Latency-Insensitive Design of a Real-World SoC. In the second set of experiments, we applied latency-insensitive design and the proposed FIC optimization to the semi-custom design of a real-world SoC 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 [16], [18]: this is a "coded orthogonal frequency division modulation" (COFDM) baseband solution for ultra-wideband systems. Fig. 10 shows a 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, (2) a latency-insensitive design (LID) version of it, and (3) a LID version with FIC optimization. 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[5], by exploiting the SD-FICs 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.

Fig. 11 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

[5] 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.

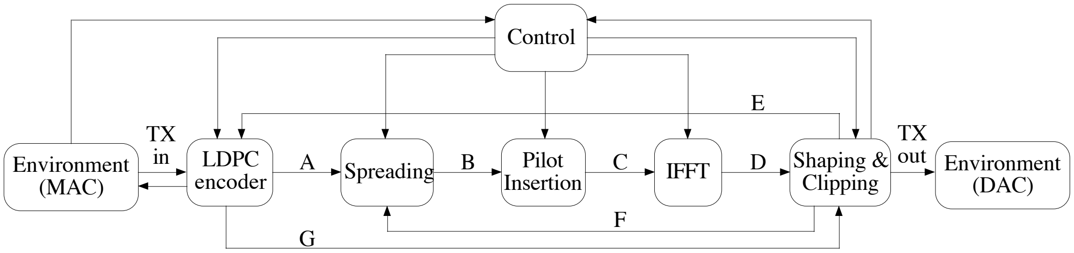


Fig. 10. The LDPC-COFDM-based ultra wideband transmitter. The channels of the datapath are labeled alphabetically.

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. The average throughput speedup across all configurations is 10.3%.

All of the FICs 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 FICs that the tool discovered are surprisingly 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 a FIC roughly depends on how often it can be used to avoid stalling of modules in the critical cycles. Fig. 12 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 being used 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 FICs 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 cycle of the design. In contrast, when a relay stations 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 cycle (which is G-E-G in this case), and channel E's FIC happens rarely and thus cannot have a sizable impact on throughput.

The ability of the proposed algorithm to discover the FICs automatically regardless of the nature of the design and without human interventions is very beneficial. For example, as explained in the COFDM publication [4], the FIC of channel B is due to the protocol design: the *Pilot-Insertion* module adds pilot symbols periodically to allow a receiver to measure the distortions of the transmitted symbols. During this operation the *Pilot-Insertion* module does not need the inputs from channel B. Our algorithm discovers this FIC automatically without the knowledge of the protocol design, and synthesizes its detection logic which is "correct-by-construction".

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), and FIC-shells with FIC-detectors add negligible area to the classical shells. The critical path delays of the classic and FIC-optimized latency-insensitive transmitters are the same as the original strict design, i.e. the maximum clock speed is not affected. This means that the effective system performance, defined as clock frequency multiplied by the system throughput [8], is often increased by applying the proposed FIC optimization over the classic latency-insensitive design.

As discussed in [12], [19], [20], 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 *cycle* 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 cycle 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 Fig. 13 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 FICs creates more opportunities for throughput optimization beyond the queue-sizing. In the example above, instead of sizing the queues, we can exploit a FIC of channel F to bring the throughput back to 0.98. This optimization 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 without FIC, or 0.79 for FIC without queue sizing). Columns labeled as "FIC" in Fig. 13 report the throughput data for the various scenarios.

VI. Conclusions

We discussed the problem of exploiting functional independence conditions on the logic design of pearl modules to optimize the performance of a latency-insensitive system. The paper's contributions include the circuit design of a FIC-shell, a logic synthesis procedure to automatically synthesize FIC-shells around pre-designed pearl modules, the experimental analysis of the benefits and overhead of the proposed technique, and, finally, its application to the synthesis of a real-world system-on-chip using the latency-insensitive design methodology.

Acknowledgments

The authors would like to thank Hsuan-Yu Liu and Chen-Yi Lee for providing the RTL design of their COFDM SoC and Claudio Pinello for useful discussions on FIC-based optimization. This research is partially supported by Intel Corporation and the GSRC Focus Center, one of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

RS locations	throughput No FIC	 FIC	speedup (%)	A's SD-FIC occurred	 used	B's SD-FIC occurred	 used	D's SD-FIC occurred	 used	E's SD-FIC occurred	 used	F's SD-FIC 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. 12. Throughput improvements with one RS insertion and shell queues of size two.

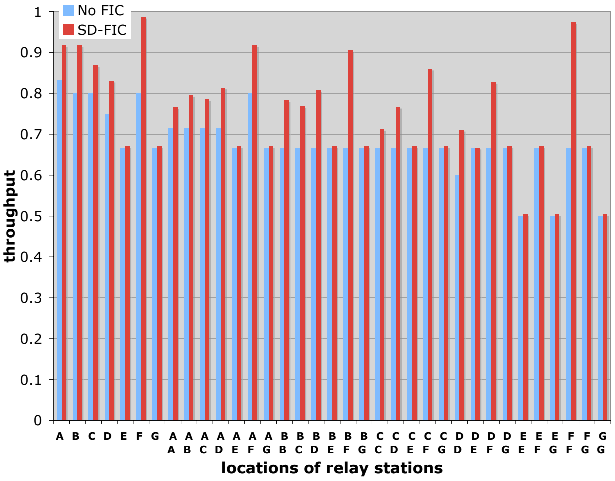


Fig. 11. Throughput improvements with one or two RS insertions on different channels.

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. 13. Impacts of the FIC optimization and queue sizing on throughputs with one RS insertion.

References

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

[2] 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*, pages 1006–1013, 2005.

[3] 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*, pages 611–618, 2006.

[4] A. Batra et al. Multi-band OFDM physical layer proposal for IEEE 802.15 task group 3a. *IEEE P802.15-03/268r1-TG3a*, Sept. 2003.

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

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

[7] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

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

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

[10] M. R. Casu and L. Macchiarulo. On-chip transparent wire pipelining. In *Proceedings International Conference on Computer Design*, pages 160–167, Oct. 2004.

[11] V. Chandra, H. Schmit, A. Xu, and L. Pileggi. A power aware system level interconnect design methodology for latency-insensitive systems. In *Proceedings International Conference on Computer-Aided Design*, pages 275–282, 2004.

[12] 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*, pages 410–416, June 2007.

[13] 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*, 16(3):72–80, 1999.

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

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

[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*, pages 1–4, 2006.

[17] 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 (MEMOCODE)*, pages 13–22, 2007.

[18] 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*, volume 1, pages 444–609, 2005.

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

[20] R. Lu and C.-K. Koh. Performance analysis of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):469–483, Mar. 2006.

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

[22] V. Nookala and S. Sapatnekar. A method for correcting the functionality of a wire-pipelined circuit. In *Proceedings of the Design Automation Conference*, pages 574–575, June 2004.

[23] 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*, pages 2–13, 2003.

[24] 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*, 24(4):532–550, Apr. 2005.

[25] 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*, pages 1008–1013, 2004.