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

Cheng–Hong Li and Luca Carloni
Department of Computer Science, Columbia University, New York, NY 10027
{cheli, luca}@cs.columbia.edu

***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 also provide an automatic procedure for the logic synthesis of a shell instance that is only based on the particular local characteristics of its corresponding pearl and does not require any input from the designers. We conclude reporting on a set of experimental results that illustrate the benefits and overhead of the proposed technique.**

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, thereby reducing the number of costly iterations in the design process [6], [8]. 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 (an FSM, a pipelined circuit,...), the only requirement is that it is *stallable*, i.e. it can be *clock gated*. Fig. 1 (from [8]) 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 desired 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 [6] 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, 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], [12]. 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 will circulate 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

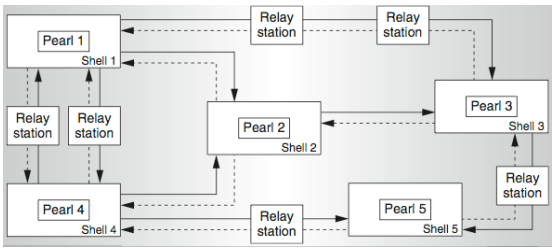


Fig. 1. Shell encapsulation, relay station insertion, and channel back-pressure.

Pearl 1 and 4 in Fig. 1 induce two bubbles circulating in the loop that stall Pearl 1 and 4 periodically, thus reducing the throughput of the entire system to 0.5. This throughput degradation can be easily computed in advance [7], [12].

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 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_1 is $\{D, E, F\}$. Since we have single-output Moore FSMs, we simply assume that in each state S the value of the output variable is equal to the corresponding lowercase letter s: in other words, FSM M_1 outputs $X = a$ while being in state A, $X = b$ while in state B, and $X = c$ while in state C. Similarly, FSM M_2 outputs $X = d$ while being in state D, $X = e$ while in state E, and $X = f$ 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 $(A \rightarrow C \rightarrow A \rightarrow A \rightarrow B) \rightarrow (A \rightarrow C \ldots$, while for M_2 we have $(D \rightarrow F \rightarrow E \rightarrow F \rightarrow E) \rightarrow (D \rightarrow F \ldots$.

[1] Notice that 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 functions for which the output value is not specified.

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

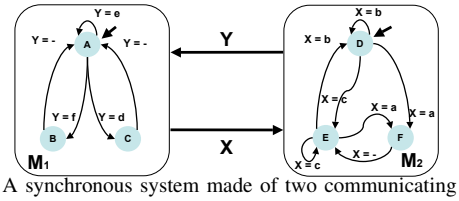


Fig. 3. A synchronous system made of two communicating FSM.

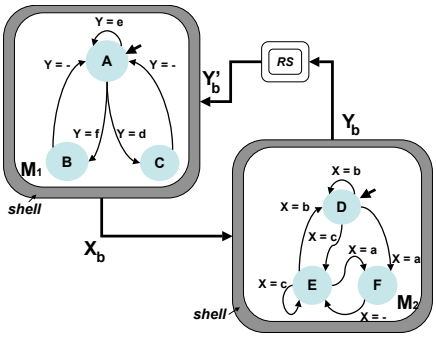


Fig. 4. A latency-insensitive system derived from the system of Fig. 3

The second set of traces 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. This value will continue to iterate forever 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 comparisons of the X, Y traces with the X_b, Y_b traces shows that they are *latency-equivalent* as expected [6]: i.e., they are the same if one ignores the τ symbols. But, the system throughput is reduced from 1 to $\frac{2}{3} = 66\%$.

Part of the lost throughput, however, can be recovered if one knows the internal structure of the FSM (an assumption not made in [6] 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: (1) avoids to stall M_2 whenever it is in state F *and* there is a τ on channel X_b (*stall avoidance*); (2) remembers that for 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 system, delaying one stall by only a single clock cycle allows us to raise the throughput by 9% to $\frac{5}{7} = 0.72$.

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 for LID systems. Our results confirm that the system performance of a latency-insensitive system can benefit considerably from this idea with reasonable 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. applies "early evaluation" to phased logic in different granularities [13], [14] while Ampalam et al. [3] and Brej et al. [4] use "anti-tokens" to support early evaluations in pipelined asynchronous logic. In this paper we essentially apply early evaluation to the optimization of *synchronous* systems in the context of the latency-insensitive design methodology [5], [6]. Specifically we start from a synchronous specification such as a network of FSMs and automatically derive a synchronous latency-insensitive implementation. The various practical advantages of starting from a synchronous specification are explained in [5], [6]. The authors of [2], [15] have proposed a related method to optimize the performance of latency-insensitive systems in the presence of multi-clock domains.

To exploit functional independence, a detection logic triggering an early evaluation must be supplied. The authors of [14] 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. 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. Since the computation of a logic component and the arrival of data tokens may mismatch, subsequent computations and new data tokens must be properly *re-aligned*. In [13], [14], this is achieved by acknowledging early and late arrival data tokens simultaneously. In [3], [4], an early evaluation generates an *anti-token* flowing in the opposite direction of normal data flows to cancel unused (and unneeded) normal tokens. In this paper, the realignment is done 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 [11]. To implement it we use simple and efficient hardware (a 1-bit shift register). In acknowledge-based realignment, 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 in our approach where back-to-back, more frequent early firings are possible. Also, while in [3], [4] modified

pipeline protocols must be adopted, our realignment method is *local* to the pearl modules and does not change the global communication protocol.

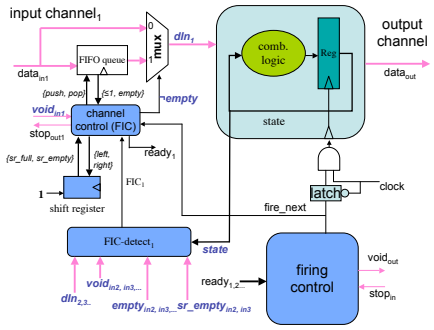
III. Shell Design

We present the design of a shell exploiting FICs (*FIC-shell*). This is a variation of the shell proposed in [5], [6], [8] 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 its 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*. A void signal is generated by the sender shell to inform the receiver whether the accompanying data is valid. The stop signal is a flow control signal and is used by a receiver to inform its sender to stop sending more data ("back-pressure") [5].

At each 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 will be discarded. Valid tokens not consumed (due to stalling) are stored in FIFO queues for later use. Thus a valid data token can come either directly from the input channel or from the FIFO. In either case, the channel is declared *ready*. The shell will also stall the pearl if a downstream receiver notifies that it cannot consume more tokens by asserting the stop signal [5].

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 as 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, this token when arrives 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 this 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 is a value that is not necessary. The count is maintained by simple rules. When a pearl is fired but the current channel presents a void token, the count is increased 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



(a)

$$
\begin{aligned}
ready_i(t) &= (\overline{void_{in_i}(t)} \cdot sr\_empty_i(t)) + \overline{\text{fifo\_empty}(t)} + \\
&\quad (FIC_i(t) \cdot \overline{sr\_full_i(t)}) \\
stop_{out_i}(t) &= 1 \text{ if empty slots in FIFO}_i(t) \leq 1 \\
push_i(t) &= sr\_empty_i \cdot \overline{void_{in_i}(t)} \cdot (\overline{fire\_next(t)} + \overline{fifo\_empty_i(t)}) \\
pop_i(t) &= \overline{fifo\_empty_i(t)} \cdot fire\_next(t) \\
sh\_left_i(t) &= \overline{void_{in_i}(t)} \cdot (\overline{FIC_i}(t) + sr\_full_i(t)) \cdot \overline{sr\_empty_i(t)} \\
sh\_right_i(t) &= fire\_next(t) \cdot void_{in_i} \cdot fifo\_empty_i(t) \cdot \overline{sr\_full_i(t)}
\end{aligned}
$$

(b)

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

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, the channel control can no longer declare a channel ready even it is not needed (a FIC). 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.[2] 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) [7], [12], FIC-shells can be used only for those pearl modules that are part of feedback loops, while classic shell 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-transfer 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 state variables together with the remaining input variables and their validity (decided by the *void* and *empty* signals). Note that our procedure *uses* observability don't cares as a starting point for FIC computation.[3]

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

[3]ODC computation is the basis of our procedure, but not the focus of this paper. For ODC computation the interested reader can refer to [9].

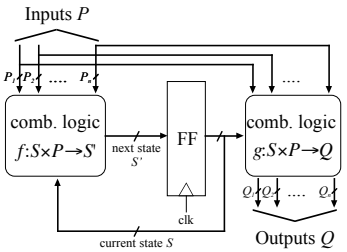


Fig. 6. Modeling of a pearl module.

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

A. Background Definitions

Without loss of generality a pearl module can be modeled as a Mealy FSM that is specified by a *state transfer 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 transfer function can be written in vector form as:

$$\mathbf{S}' = \mathbf{f}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n; \mathbf{S}) \tag{1}$$

where $\mathbf{S}' \equiv \{s'_1, s'_2, \ldots, s'_n\}$ is the vector of next state variables, $\mathbf{P}_i \equiv \{p_{i_1}, p_{i_2}, \ldots, p_{i_{|\mathbf{P}_i|}}\}$ 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

$$\mathbf{Q} = \mathbf{g}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n; \mathbf{S}) \tag{2}$$

Observability don't care. 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 [9]. This unobservability 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 $\text{ODC}_{x_i}(f)$ be the conditions under which function f is insensitive to variable x_i. We have

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

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 computed. An effective solution, which has been shown successful on large designs, is to iteratively applying Boolean difference functions locally [9]. For simplicity, in the sequel the Boolean difference will still be used as a notation to represent the computation of ODC sets.

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

$$C_{x_i}(f) = f|_{x_i=1} \cdot f|_{x_i=0} \tag{3}$$

Consensus can be extended to a set of variables by iteratively applying Eq. 3 to each variable [9].

B. 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 transfer function $f_{s'_k}$ ($s'_k \in \mathbf{S}'$ is a single next state variable). We have:

$$\widetilde{\text{ODC}}_{p_{i_j}}(f_{s'_k}) = \overline{\frac{\partial f_{s'_k}}{\partial p_{i_j}}} = f_{s'_k}|_{p_{i_j}=1} \,\overline{\oplus}\, f_{s'_k}|_{p_{i_j}=0} \tag{4}$$

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

$$\widetilde{\text{ODC}}_{p_{i_j}}(g_{q_l}) = \overline{\frac{\partial g_{q_l}}{\partial p_{i_j}}} = g_{q_l}|_{p_{i_j}=1} \,\overline{\oplus}\, g_{q_l}|_{p_{i_j}=0} \tag{5}$$

Step 2. Since FICs involve all state and output variables we perform the conjunction of all the FICs computed by Eq. (4) and Eq. (5):

$$\widetilde{\text{ODC}}_{p_{i_j}}(\mathbf{f}, \mathbf{g}) = \Big(\bigwedge_{s'_k \in \mathbf{S}'} \text{ODC}_{p_{i_j}}(f_{s'_k}) \Big) \cdot \Big(\bigwedge_{q_l \in \mathbf{Q}} \text{ODC}_{p_{i_j}}(g_{q_l}) \Big) \tag{6}$$

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:

$$\begin{aligned} \widetilde{\text{ODC}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g}) &= C_{\mathbf{P}_i}\Big(\bigwedge_{p_{i_j} \in \mathbf{P}_i} \text{ODC}_{p_{i_j}}(\mathbf{f}, \mathbf{g}) \Big) \\ &= C_{p_{i_1}}\big(C_{p_{i_2}}(\cdots C_{p_{i_j}}\Big(\bigwedge_{p_{i_j} \in \mathbf{P}_i} \text{ODC}_{p_{i_j}}(\mathbf{f}, \mathbf{g})\Big)\cdots\big) \end{aligned} \tag{7}$$

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. (7) to come from input channels presenting valid tokens. Recall that a good token can be either from the channel (i.e. its $void$ is 0) or from the channel's FIFO queue (i.e. the FIFO is not empty). Further, if the token is from the channel, it cannot be outdated (shift register must be empty). So the final FIC conditions can be obtained as follows:

$$\begin{aligned} \text{ODC}^{\text{IS}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g}) \equiv \text{Replace each literal } p \text{ in } \widetilde{\text{ODC}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g}) \text{ with} \\ p \cdot (\overline{void_k} \cdot sr\_empty_k + \overline{empty_k}), \text{ and } \overline{p} \\ \text{with } \overline{p} \cdot (\overline{void_k} \cdot sr\_empty_k + \overline{empty_k}) \end{aligned} \tag{8}$$

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

The domain of the single-output Boolean function $\text{ODC}^{\text{IS}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g})$ that is obtained at the end of Step 4 is the set of state variables, input variables, $void$ and $empty$ variables minus the set of input, $void$ and $empty$ 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 $\text{ODC}^{\text{IS}}_{\mathbf{P}_i}(\mathbf{f}, \mathbf{g}) = 1$ then the current data value of channel $\mathbf{P}_i$ is not needed to compute the state and output function of the pearl.

FIC conditions that depend on input channels may induce extra timing constraints. In fact, the firing of a pearl module is controlled by signal *fire_next* signal, which must be stable by the end of each clock cycle. The dependency of FIC conditions 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

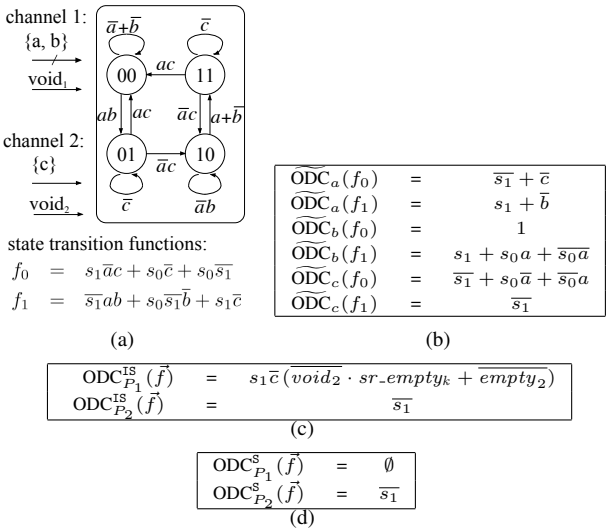


Fig. 7. (a) A pearl module with 2 input channels (3 input variables in total) modeled by a 4-state Moore FSM. (b) The ODC sets of each input variable with respect to the 2-state transfer functions. (c) The final FICs depending both on inputs and states. (d) The final FIC depending only on states.

want to restrict ourselves to FIC depending only on state variables. This requires a different (alternative) final step in our procedure.

Step 4'. To restrict FIC conditions to state variables only, we apply the consensus function to Eq. (7) over all input variables iteratively:

$$\begin{aligned}\text{ODC}^{\text{S}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g}) &= C_{\mathbf{P}}(\widetilde{\text{ODC}}_{p_i}(\mathbf{f},\mathbf{g})) \\ &= C_{p_{1_1}}(C_{p_{1_2}}(\cdots C_{p_{i_j}}(\widetilde{\text{ODC}}_{p_i}(\mathbf{f},\mathbf{g})\cdots))) \end{aligned} \quad (9)$$

If the pearl module has no combinational path from its inputs to outputs (thus it can be viewed as a Moore FSM), Eq. (5) will return 1 because an output variable does not depend on any input. The same steps can be applied to compute on a channel basis thereby $\text{ODC}^{\text{S}}_{\mathbf{P}_i}(\mathbf{f},\mathbf{g})$ is simply $\text{ODC}^{\text{S}}_{\mathbf{P}_i}(\mathbf{f})$.

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 transfer 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 ($s_0s_1 = \{00, 01, 10, 11\}$).

We applied our four-step procedures to derive the FICs for each input channel. Since the pearl is a Moore FSM, only Eq. (4) 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. (6) and Eq. (7) provide the FIC for each of the two channels: $\text{ODC}^{\text{IS}}_{\mathbf{P}_1}(\mathbf{f}) = s_1\overline{c}\,(\overline{void_2} \cdot sr\_empty_k + \overline{empty_2})$ and $\text{ODC}^{\text{IS}}_{\mathbf{P}_2}(\mathbf{f}) = \overline{s_1}$.

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 $\text{ODC}^{\text{S}}_{\mathbf{P}_2}(\mathbf{f}) = \overline{s_1}$, while the input data coming at channel 1 are always needed: $\text{ODC}^{\text{S}}_{\mathbf{P}_1}(\mathbf{f}) = \emptyset$. 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

This section presents various experiments designed to evaluate the applicability and efficiency of the proposed optimization technique. We implemented the procedure discussed in Section IV within the logic synthesis tool ABC [1] and we test it with the ISCAS-89 benchmark suite and other sequential circuits. For each benchmark, the functional independence conditions (FIC) are derived assuming that each single input is a LID channel. 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 frequencies of FIC in reachable states for benchmark $s1488$: 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 $s1488$, SD-FIC conditions are very frequent: all but two inputs are SD-FIC in most states. Further, in most reachable states, there is a significant number of FIC inputs. Note that SD-FIC dominates, and by considering ISD-FIC only a little more FIC conditions can be exploited.

Table 9 shows the results of measurements of FIC frequencies across all benchmarks. For each benchmark, the column "# of SD-FIC inputs" reports the number of inputs which is a 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 in the following column. The same analysis is applied to ISD-FIC conditions, and results are listed in the last three columns. *The results indicate that FIC conditions 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 $s349$. This is an add-shift-multiplier [10], controlled by a 3-bit counter. Its inputs are only needed in the first cycle of each computation round (thus state-dependent).

Table 10 shows the area and delay overheads of FIC-detect blocks. All of the FIC-detect blocks and benchmark circuits are synthesized and mapped using ABC's synthesis scripts. For FIC-detect blocks, both area and delay are reported in absolute values and in ratios compared to the core components'. A dash ("-") entry indicates there is no SD-FIC (or ISD-FIC) condition for any input of the benchmark. In most cases the logic detecting ISD-FIC are larger and slower than the logic detecting SD-FIC. In some cases the ISD-FIC detect logic is even bigger and slower than its core, which makes ISD-FIC conditions infeasible in practice. On the other hand, SD-FIC detect logic imposes much lower costs. Note that in either types of detect logic, as long as its delay is smaller than the delay of the pearl's critical path (i.e. ratio smaller than 1), the detect logic will not increase the clock cycle time. This assumes that the FIC-detect logic is not on the critical path, a reasonable assumption given the simple logic driven by FIC-detect in our shell design.

These results confirm that in practice it is sufficient to focus on exploiting SD-FIC since they already offer many opportunities to improve the performance of a latency-insensitive system. Further, SD-FIC have a much more limited area overhead than ISD-FIC and do not introduce any delay penalty.

VI. Conclusion and Future Work

We discussed the problem of exploiting functional independence conditions on the logic 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 a FIC-shell, and the first analysis of the benefits and overhead of the proposed technique that is based on experimental results. Future work will include the application of this idea to a real SOC consisting of a network of multiple pearls.

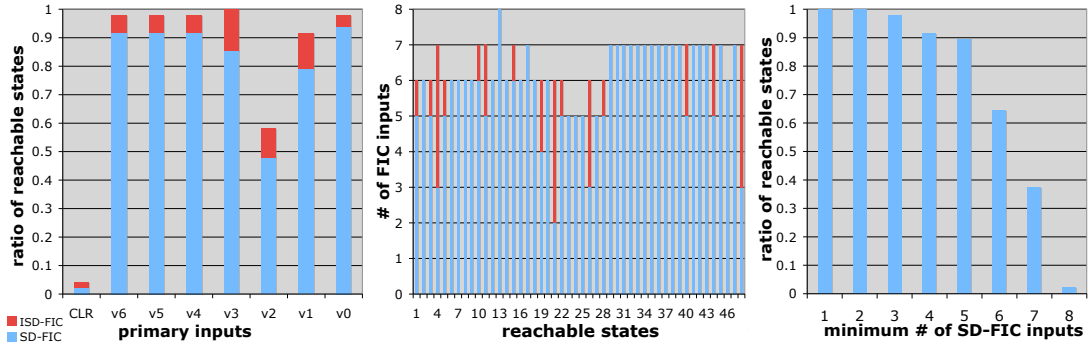


Fig. 8. Frequency distributions of FIC conditions in s1488. In Fig. 8–10, the acronym "ISD-FIC" refers to functional independence conditions depending on state variables and at least one input variable; "SD-FIC" refers to conditoins depending only on state variables.

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 of FIC frequencies across all benchmarks.

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 *ICCAD*, pages 1006–1013, 2005.

[3] M. Ampalam and M. Singh. Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens. In *ICCAD*, pages 611–618, 2006.

[4] C. F. Brej and J. D. Garside. Early output logic using anti-tokens. In *IWLS*, pages 302–309, 2003.

[5] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for "correct-by-construction" latency insensitive design. In *ICCAD*, pages 309–315. IEEE, Nov. 1999.

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

[7] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *DAC*, pages 361–367.

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

[9] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. 1994.

[10] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: a case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[11] J. Júlvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *ICCAD*, 2006.

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

[13] R. R. Reese, M. A. Thornton, and C. Traver. A coarse-grain phased logic CPU. In *ASYNC*, pages 2–13, 2003.

[14] R. R. Reese, M. A. Thornton, C. Traver, and D. Hemmendinger. Early evaluation for performance enhancement in phased logic. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(4):532–550, Apr. 2005.

[15] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *DATE*, pages 21008–21013, 2004.

Bench	Core		SD-FIC Detect		ISD-FIC Detect	
	Area	Delay	Area (%)	Delay (%)	Area (%)	Delay (%)
s1488	1048	10.30	165 (16)	6.9 (67)	322 (31)	8.6 (83)
s208	116	6.20	38 (33)	3.7 (60)	118 (102)	8.8 (142)
s27	21	4.50	7 (33)	1.9 (42)	42 (200)	5.2 (116)
s298	194	7.00	- -	- -	48 (25)	6.0 (86)
s349	220	7.80	44 (20)	4.6 (59)	44 (20)	4.6 (59)
s382	241	7.30	- -	- -	52 (22)	5.8 (79)
s386	226	8.70	57 (25)	5.5 (63)	154 (68)	9.2 (106)
s510	407	7.40	117 (29)	3.7 (50)	190 (47)	6.2 (84)
s526n	351	7.20	- -	- -	87 (25)	7.0 (97)
s832	491	9.00	139 (28)	6.0 (67)	430 (88)	8.3 (92)
s953	710	8.20	107 (15)	5.5 (67)	1081 (152)	13.9 (170)
ex1	436	7.90	67 (15)	5.0 (63)	162 (37)	7.2 (91)
keyb	463	9.30	72 (16)	4.5 (48)	620 (134)	11.5 (124)
kirkman	282	7.80	33 (12)	3.5 (45)	711 (252)	11.6 (149)
planet1	1266	9.30	146 (12)	6.6 (71)	327 (26)	9.9 (106)
sand	1128	9.80	89 (8)	5.9 (60)	402 (36)	10.6 (108)
shiftreg	12	1.90	- -	- -	- -	- -
Add256Cntrl	93	5.40	12 (13)	3.6 (67)	12 (13)	3.6 (67)
boltzmann	524	8.20	73 (14)	6.3 (77)	92 (18)	6.3 (77)
lan	293	10.00	38 (13)	3.9 (39)	307 (105)	10.1 (101)
TagGen	144	4.80	- -	- -	8 (6)	1.6 (33)
TagGenCntrl	97	4.90	21 (22)	3.6 (73)	27 (28)	4.0 (82)
Avg.	398	7.40	72.1 (19)	4.7 (60)	249.3 (68)	7.6 (98)

Fig. 10. Overheads of FIC-detect logic in terms of area and delay.