2018 International Conference on Field-Programmable Logic and Applications

Latency Insensitive Design Styles for FPGAs

Mustafa Abbas and Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
{mabbas, vaughn}@ece.utoronto.ca

***Abstract*—Long distance interconnect delays are not scaling well with process technology, thereby leading to long routes strongly impacting the critical path of large FPGA designs. This forces the designer to pipeline long connections, which necessitates time consuming logic redesign in traditional latency-sensitive systems. Latency-insensitive design (LID) is an increasingly attractive alternative as the typical latency of long distance interconnect grows, since LID decouples the design of the interconnect from that of the computational modules. By doing so, LID simplifies timing closure, improves forward compatibility (migration of systems to future FPGAs) and makes automated system-level pipelining feasible. Modern FPGAs, such as Stratix 10 which includes pipelined interconnect, make it difficult to use traditional LID solutions without significant area and frequency overhead. We present two LID styles that are more suitable for FPGAs and compare them to traditional LID. Our best system gained 2x area efficiency and 18% speed efficiency over traditional LID. Additionally, our designs come at a minimal speed overhead of only 3% compared to that of a latency-sensitive design.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) continue to scale in capacity with the introduction of every new process generation. For example, the recently released Stratix 10 FPGA has as many as 5-million Logic Elements on a chip; this is 5x greater than the amount present in Intel's previous generation Arria 10 FPGA [1], [2]. While transistor density on a chip continues to increase, wire speed for the same length of wire is not scaling [3]. Thus, global connections that traverse a large area of the chip strongly impact the critical path of large designs. In fact, over five Stratix FPGA process generations, from 130 nm to 28 nm, interconnect that spans the length of the chip had no speed gain [4]. The same trend continues with the 14 nm Stratix 10, which like the previous generation FPGA achieves ∼100 MHz register to register operating frequency for connections spanning the chip according to our experiments. This global wiring scaling problem exacerbates one of the major challenges facing hardware designers today, Timing Closure: ensuring the design meets all its timing requirements.

Currently, a designer using the common synchronous design methodology must perform many costly iterations of the entire FPGA CAD flow (synthesis, place, and route) to reach timing closure. First, the designer must manually add pipeline registers to critical paths of the design and update design logic to maintain functionality given these modifications. Then they must re-run the CAD flow and verify that the design was able to meet timing. The designer has to iterate these two steps until the design has reached timing closure while remaining functionally correct. This is a time consuming task, since running through the CAD flow requires multi-hour-to-multi-day runtimes [5]. In addition, the entire problem is reintroduced if the designer chooses to move their design to a new FPGA chip, as a different FPGA architecture and process technology affects the timing of the design. Hence, even if the designer has validated the IPs in their design on the new FPGA, the designer must tediously fine tune the interconnect to meet timing again.

Recent demand for FPGAs in the datacenter have put pressure on designers to be more productive [6]. In the data center IP cores will be combined into accelerators in more diverse ways. This adds more motivation to move to a methodology with effective forward compatibility that can reach timing closure with low effort.

One way to simplify timing closure and IP reuse for the system level, is for the designer to use a latency insensitive design (LID) methodology [7]. LID separates the design of the interconnect from the computational modules. By doing so it allows the designer to add as many pipeline stages in the interconnect as needed to meet the desired clock frequency without breaking the functionality of the design. Additionally, it makes it possible to develop CAD tools that can automatically pipeline the interconnect later in the CAD flow, thereby reducing time-consuming iterations of the design.

Generally LID has been based on a design style introduced by Carloni [8] which uses a structure called a Relay Station for pipelining. Work in [4] quantified the cost and benefits of using Carloni-based LID on FPGAs. It concluded that the area and timing overhead is manageable, given the benefits LID provides for connecting large designs in FPGAs. We note however that this design style was developed for ASICs and the cost of circuitry on FPGAs is significantly different. Registers and multiplexers on the FPGA are relatively expensive and block-RAMs are plentiful making solutions that make heavier use of FIFOs more attractive. Recently, there has been a major change in FPGA architecture with Stratix 10, which includes registers (hyper-registers) within the FPGA routing interconnect. The commercial introduction of hardened interconnect registers increases gains for deep pipelining by using the less costly interconnect registers for pipelining, making it a good fit for LID. However, hyper-registers do not have direct access to enable or clear ports, motivating new LID styles that can efficiently exploit such registers [9]. We develop two different LID styles that can easily target hyper-registers on communication links. We compare them to Carloni-based LID which requires more advanced pipelining structures with wide multiplexers and high-fanout enable signals to implement Relay Stations. Our comparison is performed on both Arria 10, a more traditional FPGA architecture, and Stratix 10 which includes hardened interconnect registers.

978-1-5386-8517-4/18/$31.00 ©2018 IEEE
DOI 10.1109/FPL.2018.00068

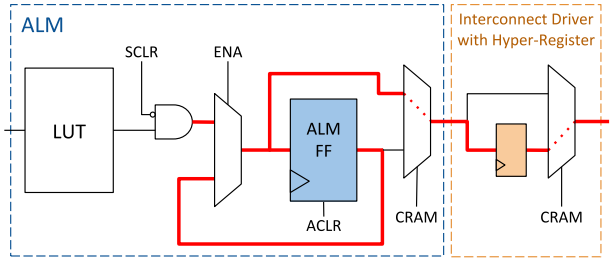

Fig. 1: ALM Control Signals [9]; highlighted path (red) shows the use of register control signals with a hyper-register

II. Registered Routing in Stratix 10

In this section we detail pipelined interconnect used in a commercially-available FPGA, providing background for later parts of the paper where we explore different latency insensitive design styles that better target new FPGAs.

The Stratix 10 pipelined architecture was introduced to mitigate the impact of long range routing delays and to be well suited for high-speed pipelined designs. This was done by adding a bypassable hyper-register to every routing multiplexer in the FPGA [9]. Hyper-registers enable the migration of a register in a design to anywhere on the interconnect, allowing for greater flexibility in pipelining long connections.

A hyper-register is implemented as a pulse latch which logically acts as an edge triggered flip-flop (FF). Unlike a conventional FF, it has no direct access to control signals like clock enable and clear ports [9]. Therefore, hyper-registers are most easily used for simple registers in a design that do not need these control signals. Registers in a design that use clock enable (ENA) or synchronous clear (SCLR) can still benefit from hyper-registers, as shown in Fig. 1. An ALM register is used to implement the ENA and/or SCLR functionality, and a simple register duplicate is implemented as a hyper-register by appropriate configuration-RAM (CRAM) cell settings. This can improve timing for registers with ENA or SCLR, but cannot reduce area by saving ALM registers. As for the asynchronous-clear (ACLR) port, it cannot be used together with the the hyper-register as seen in Fig. 1.

In the next section we introduce two latency-insensitive systems that can utilize hyper-registers efficiently as they do not use any clock enable ports or synchronous clears in their pipeline elements. Thereby they are able to save area and delay by not requiring passage through an ALM for pipelining. We also detail Carloni-based LID that we use for comparison.

III. Three Different Latency Insensitive Styles

Latency Insensitive Design (LID) is the process of assembling design modules using communication links that are allowed to transfer data at arbitrary latencies. Each design module's only guarantee is the synchronization on communication channels and order of the arriving and sent data. This allows the insertion of additional pipeline stages on latency insensitive links, at any stage in the CAD flow, without changing the functionality of the design [7].

The benefit of LID is that it allows the hardware engineer to design their system in two separate stages:

- *Computation:* The design of the IP cores' functionality.
- *Communication:* The interconnect architecture used for communication between computational cores.

The separation of design of the module functionality is done via shell encapsulation of the module (usually referred to as the pearl) using a latency insensitive wrapper. The wrapper can be automatically generated and is comprised of buffering stages and control logic used to synchronize data on communication channels as well as to control stalling of the pearl module using a backpressure signal. The only requirement of the pearl module is that it is stallable.

For a module to be stallable it must be able to freeze input and output data in the event of a stall. This is a simple modification and can be implemented by sending a clock enable signal to all flip-flops inside the pearl module as implemented in [4]. The wrappers in turn make the module *patient*. A patient module is one that behaves correctly regardless of the interconnect latency. The resulting design greatly simplifies the interconnect architecture allowing additional pipelining of communication wires between patient modules [7]. We study the use of three different approaches to LID that utilize different pipeline elements and wrappers. However, the abstraction all three LID approaches present is the same such that they are equally easy to integrate into designs.

A. Carloni Based Latency Insensitive Design

Carloni and Vincentelli introduced a wrapper and pipelining element design for LID in [10]. Variations of their designs can be seen in [4], [11]. We adapted the original work with the addition of input port buffering queues for the wrapper data channels. Although the buffering queues are not necessary for the functionality of the system, they can optimize throughput in systems where wrappers have multiple input channels [12]. The wrapper design can be seen in Fig. 2, and the pipelining element called a Relay Station can be seen in Fig. 3 along with its control finite state machine (FSM) in Fig. 4.

The shell interface consist of data signals, a valid signal, and a backpressure stop signal for each channel of communication. The backpressure stop signal as well as the state of the FIFO determine when to enable the pearl module. If the pearl is not taking in valid data, i.e. the FIFO is empty or a stop signal has been raised from upstream, the downstream wrapper will send out invalid data, which is referred to as a void token.

The Relay Station is also patient; it allows for the use of two word deep buffers for pipelining the communication between shells. In the event of a stall the current data is stored in the main register bank along with its valid bit and immediately the output data is voided using the multiplexer in front of the main valid register bank as seen in Fig. 3. Any incoming value is stored in the auxiliary register bank along with its valid bit. When exiting a stall the output value is read from the main register bank, then the next value read is from the auxiliary register bank provided there is no additional stall.

The progression of these events can be seen in the Relay Station control FSM in Fig. 4. There are four total states. The

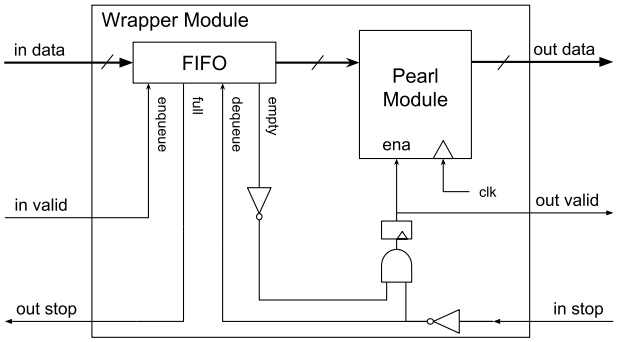

Fig. 2: Carloni based LI shell encapsulation

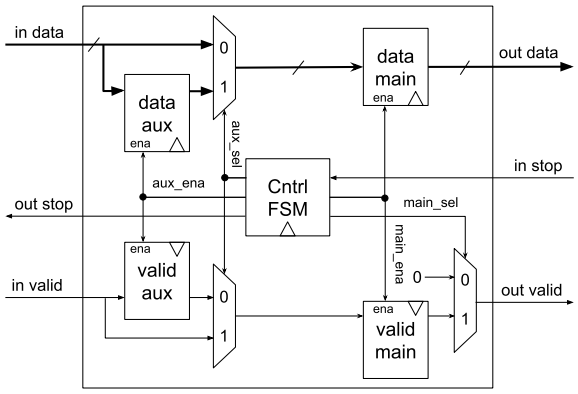

Fig. 3: Relay Station Design used for Carloni LID pipelining

Process state allows data to go through the relay station. The *Stall* state stops data and transmits void packets instead. *Write-Aux* and *Read-Aux* states guide the transitions from Process and Stall by storing data in auxiliary registers in the event of a stall and reading data from auxiliary registers when exiting a stall. All registers in the design are controlled through clock enable signals, and selecting between auxiliary or main storage is controlled using multiplexer select signals seen in Fig. 3.

B. Ready-Valid Based Latency Insensitive Design

The ready-valid system is analogous to the request-acknowledge handshake signals that are adapted as an open standard in AXI4 and Avalon interconnect [13], [14]. The wrapper design can be seen in Fig. 5. This design uses FIFO queues to buffer data in the event of a stall. The FIFO size determines the number of pipeline registers. As many pipeline stages as half of the amount of words in the FIFO can be added to the interconnect. This means that unlike the Carloni-based LID system, the FIFO is necessary and must have a depth of at least 2x the number of pipeline stages between communicating channels for correct functionality. If the FIFO is empty, then the module is stalled and the wrapper outputs void data. On the other hand, if the FIFO is almost-full, with the value of almost-full set as the round trip latency of pipelined interconnect subtracted from the number of words in the FIFO, a downstream active low ready signal is sent for back-pressure stalling.

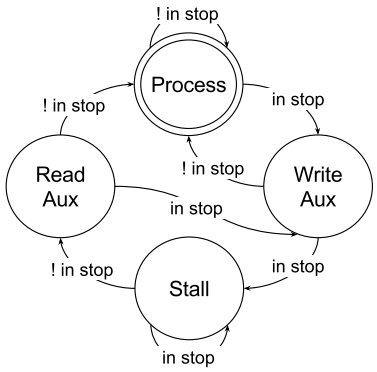

Fig. 4: Relay Station FSM used for Carloni LID pipelining

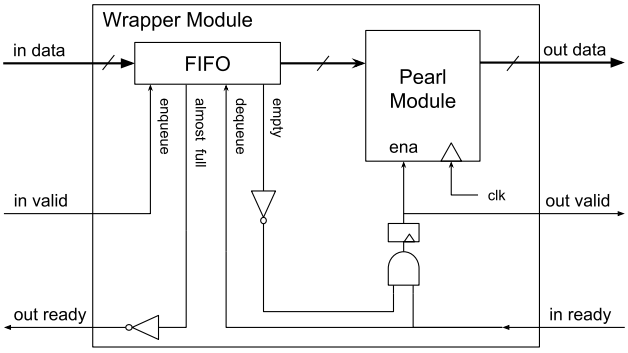

Fig. 5: Ready-Valid based LI shell encapsulation

One major advantage over the Carloni-based system is that we can use regular registers instead of relay stations which allows it to easily target the hyper-registers in Stratix 10. This is because the ready-valid FIFO is required to have enough space to store the contents of the pipeline in case of a stall without the need for distributed storage elements, such as relay stations, to perform this action.

C. Credit-Based Latency Insensitive Design

Networks-on-Chip frequently use a credit-system flow control method to avoid data loss [15] and thus can be considered as a form of LID. We develop a simple system based on the same ideas of credit-based flow control used in NoCs as a latency insensitive wrapper as shown in Fig. 6. The interface control signals are a valid bit and an increment count bit. A counter in each wrapper is used to store the number of words available in the upstream FIFO, called credits. If the counter runs out of credits then the pearl is stalled to avoid sending valid data upstream. A simple increment counter module is also placed in each wrapper to let the downstream module know that it can send more data by updating the downstream credit every time a word is dequeued from the FIFO buffer. In order to maintain functionality, the number of words in the FIFO must be greater than or equal to the round-trip latency (2x number of interconnect channel pipeline stages in forward and backpressure path) between communicating modules.

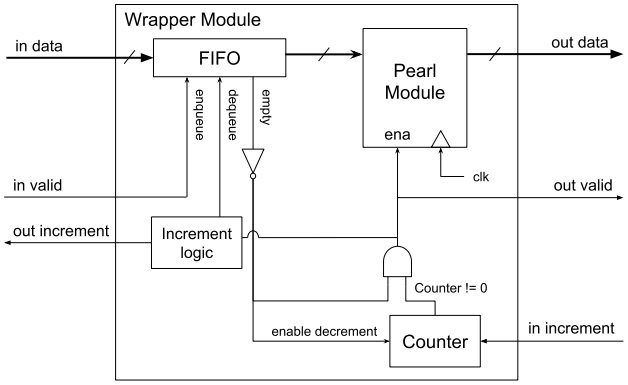

Fig. 6: Credit based LI shell encapsulation

The credit-based system can also use conventional pipeline registers instead of relay stations. It also avoids the use of enable signals, allowing it to easily target hyper-registers.

IV. Area Results

The following sections compare the area and speed of the three different LID styles in two FPGA families: Arria 10, which has traditional interconnect and Stratix 10 which has optional pipeline registers in its interconnect.

We gathered results using the Quartus Prime Pro 17.1 CAD tool. The target FPGAs used in our experiments are the Arria 10 10AS066H1F34E1SE, and the Stratix 10 101SX210HU3F50E1VG. Both are mid-size FPGAs and operate at the fastest speed grade. All reported results are averaged over 5 seeds to minimize noise due to the stochastic nature of CAD algorithms.

To compare the total layout area between LIDs, which use hardened RAM blocks, we summarize their area in terms of equivalent ALMs (eALMs). In [16] it is reported that cost of a M20K RAM block is 40 eALMs for the Stratix V architecture. Since Arria 10 and Stratix 10 use the same M20K block and similar ALM design to Stratix V we assume the ratio stays the same. In addition, when reporting Stratix 10 results we assume the use of hyper-registers does not add any area. This is because they are plentiful in the chip and always available along the current routing path. Hence, logic and RAM utilization are what will limit implementable design size.

For optimal hyper-register usage in the Stratix 10 designs we followed the suggestions in [17] including correctly setting Quartus CAD assignments and using their verilog templates.

A. Wrapper Area

Table I shows the relative area of the latency-insensitive wrappers for Stratix 10 and Arria 10 respectively. The reported resources are ALUTs, flip-flops (FF), and M20K blocks. ALUTs and FFs are merged into ALMs; finally ALMs and M20K blocks values are used to calculate the final eALM result. To gather the area results we used a simple pearl module consisting of a registered 32-bit wide datapath. The results shown in the tables exclude the area of the pearl module and show only the area overhead of a single wrapper. In Table I we see that the area overhead of all the latency intensive wrappers is relatively small, less than 70 eALMs.

Additionally, the area of the wrappers does not increase with interconnect pipelining depth. The M20K Block RAM (BRAM) can store 512 words. For the credit-based and ready-valid wrappers that require the buffering depth to be at least twice the number of interconnect channel pipeline stages, this means that you can add as many as 256 pipeline elements without the need for additional BRAM. This is many more than needed to cross even the largest FPGAs at high frequency. The only other possible area increase due to pipelining is the size of the credit counter for the credit-based wrapper, which is relatively small. The area of the credit-based wrappers reported in Table I include a 7-bit counter for the Stratix 10 design and a 5-bit counter for the Arria 10 design. These are sufficient for 128 pipeline stages and 32 pipeline stages respectively. Their area overhead is small, $\sim$6 ALMs for both cases.

Although the wrappers can handle more pipeline stages with no increase in area, this is not the case for increasing the port width. While the pearl control logic does not grow with datapath width, the FIFO width does. The M20K BRAM can handle a maximum of 40-bit wide words. Increasing the datapath beyond 40-bits requires the addition of extra BRAM blocks. Since the area of our wrappers is mostly dominated by the area of M20K blocks this can become expensive. For example, having a 64-bit datapath would require two BRAMs, nearly doubling the area of the LI wrappers. If we move to a very large datapath of 512-bits we would require a total of 13 BRAMs and increase the wrapper area to 590 eALMs which is 8.4x the size of the LI wrappers at 32-bits.

Increasing the number of input channels also increases the overhead of the wrappers due to the addition of the extra BRAM blocks needed for each input. A Small amount of control logic for the FIFO and pearl enable is also added with each extra input but this area increase is trivial compared to the area of the BRAM blocks.

B. Pipeline Area

To fully compare the LID styles we consider not only the area of the module wrappers but also the area of the pipeline stages between them. For both Stratix 10 and Arria 10 we compare the area of adding 10 pipeline elements between two modules floorplaned to opposite corners of the FPGA.

Table II shows the area of pipelining for Stratix 10 and Arria 10 respectively. Recall that the area cost is determined by the number of ALMs used because we assume hyper-registers are plentiful thus they are free. The first big trend we see is that pipelining is cheaper for credit-based and ready-valid systems compared to the Carloni system. The Carloni system requires 1.9x the area for pipeline elements compared to other LI solutions for the Arria 10 FPGA. This is because relay stations contain 2x the registers verses the other systems and also need additional area for multiplexers and control signals. The area gap is much greater on the Stratix 10 FPGA at 3x. Compared to non-LI, the credit-based and ready-valid system have a small area overhead of less than one equivalent LAB.

TABLE I: Area overhead of 32-bit wide latency intensive design wrappers

Resource	Stratix 10 Designs			Arria 10 Designs		
	Carloni	Credit	Ready-Valid	Carloni	Credit	Ready-Valid
ALMs (ALUT + FF pairs)	23	29	23	14	20	15
ALUT	40	54	41	32	41	32
FF	34	42	34	26	32	26
M20K	1	1	1	1	1	1
Equivalent ALMs	63	69	63	54	60	55

TABLE II: Area of adding ten 32-bit wide pipeline stages for each method of LID as well as a non-LI implementation

Resource	Stratix 10 Designs				Arria 10 Designs			
	Carloni	Credit	Ready-Valid	Non-LI	Carloni	Credit	Ready-Valid	Non-LI
ALMs	327	90	98	78	338	173	172	157
ALUT	405	0	0	0	71	0	0	0
FF	588	175	200	164	717	341	340	320
Hyper-registers	264	165	162	156	–	–	–	–
Equivalent ALMs	327	90	98	78	338	173	172	157

This is because the feedback signal in the LI systems is also pipelined.

We also see that pipelining is cheaper for all systems, except the Carloni relay stations, for Stratix 10 than it is in Arria 10. The area decrease due to the availability of plentiful hyper-registers in Stratix 10 is significant: ∼50 % of the eALM area. The Carloni-based system still uses a significant number of hyper-registers but this is done purely for timing. It does not save any area because the registers inside the relay stations use an enable signal which means that an ALM duplicate register is needed as shown in Sec. II.

We expected that when pipelining is all simple registers the pipelining area would be near zero. Since the interconnect for the credit-based, ready-valid, and non-LI systems is made up of only simple registers with no enables, they should mostly be targeting hyper-registers with minimal ALM flip-flops. However, we found that Quartus typically only used 2-3 hyper-registers in series before going through an ALM register, perhaps to avoid hold time violations that may arise from chaining closely spaced pulsed latches.

V. Speed Results

A. Pipeline Scaling

To study the speed of the LI wrappers we start off by analyzing the efficiency of communication over long distances using the previously mentioned 32-bit wide simple circuit, floorplanned to opposite corners of the FPGA. For the Stratix 10 design we added a phase-locked loop (PLL) with a 1 GHz output clock to avoid having clock pin I/O delays limit the critical path. Fig. 7 shows the operating frequency of the design at various levels of pipeline depth. A surprising result is the substantial number of pipeline stages needed for full frequency communication across the chip in Stratix 10 compared to Arria 10. Sixty pipeline stages are needed to operate at the best achieved frequency of 750 MHz across the chip. This is 6x the number needed for Arria 10 highlighting the increased importance of high-latency communication as process technology scales.

In Fig. 7 we can also see that the credit-based and ready-valid system achieve the same frequencies as the non-LI system. Whereas previous work showed that there is a speed overhead of 17% for LI systems on FPGA [4], using credit-based and ready-valid systems we were able to close the speed gap. This is because they use the same type of interconnect pipelining as the non-LI system and the wrappers are optimized for speed. On the other hand, the Carloni systems' operating frequency plateaus before all the other systems and is is clearly inferior for high levels of pipelining.

The average Fmax difference of Carloni-based LID verses the FIFO-buffering-based LI solutions (ready-valid and credit-based) across all pipelining depths is 60 MHz for the Arria 10 and 90 MHz for the Stratix 10. The best Fmax difference is even greater, 66 MHz on the Arria 10 and 164 MHz on the Stratix 10. The bigger difference on the Stratix 10 verses Arria 10 can be attributed to Stratix 10's hyper-registers in the interconnect which are better exploited by the credit-based and ready-valid wrappers.

B. Width Scaling

We also studied the effect of port data widths on speed. To do this we again used the the previously mentioned 32-bit wide simple circuit, floorplanned to opposite corners of the FPGA. We varied the width of the input and output channels of the wrapper/pearl module pair. For each width we averaged the speed result across multiple levels of pipelining from 0-100 pipeline stages. Fig. 8 shows this result.

As the data width increases, the speed of all systems decrease. The credit-based and ready-valid system frequencies decrease slightly more than the non-li system. This is because the critical path is increasingly within the LI wrappers due to the higher fanout FIFO control signals for high widths. The Carloni-based system's frequency is significantly lower than those of the other systems at all widths and also decreases with width at about the same rate as the other LI systems. The Carloni-based system's critical path though is due to the high-fanout control signals inside the relay stations which are proportional to the change in width. The high-fanout control

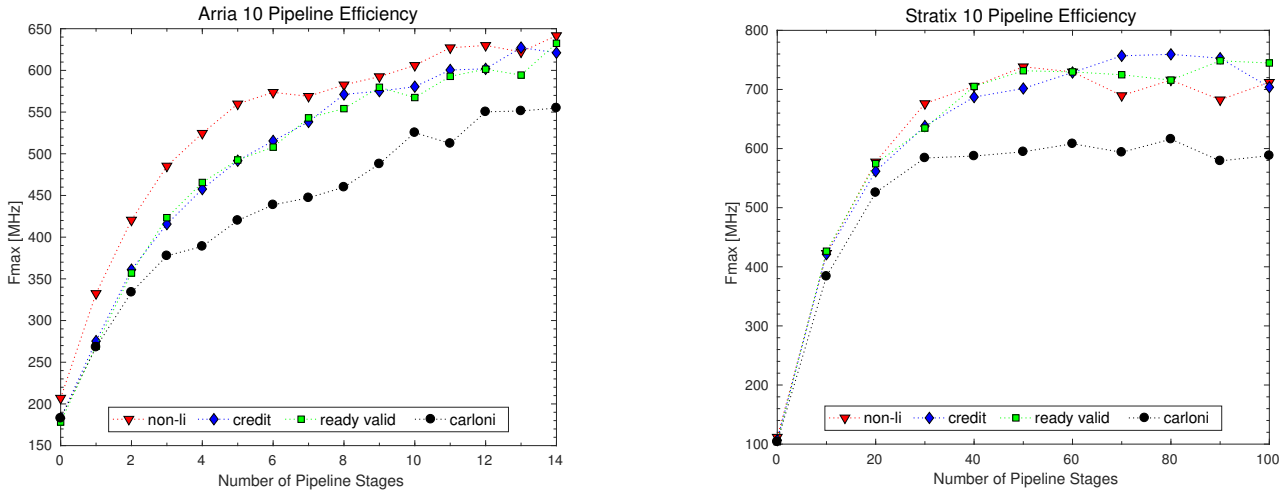

Fig. 7: Average operating frequency across five seeds for corner-to-corner communication as the amount of pipelining increases.

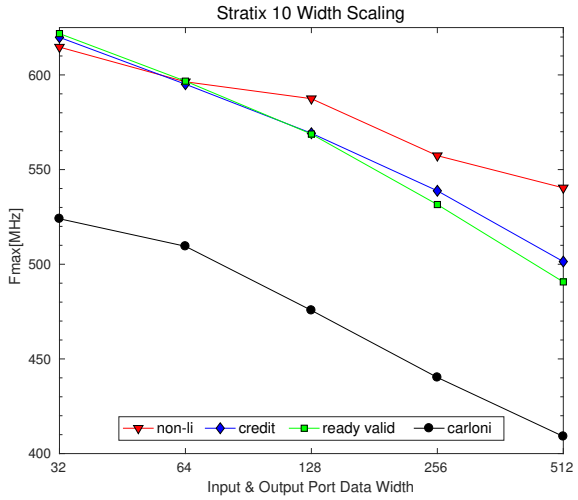

Fig. 8: Average operating frequency across 10 different pipeline depths ranging from 0-100 as port size increases.

TABLE III: Average operating frequency for different design styles when varying pipeline depths and data-widths

Design	Fmax [MHz]	Difference to Non-LI
Non-LI	579	–
Credit	565	-2.4%
Ready-Valid	562	-2.9%
Carloni	471	-20.6%

signals of the relay-station grow in delay as the data width increases due to the additional signals needed to control an increasing amount of registers in the relay station.

We average the frequency data of Fig. 8 across the various port data widths to obtain the overall frequency comparison of the four design styles shown in Table III. Overall the speed reduction for credit-based and ready-valid systems is minimal at 3% slower than the non-li system, while the Carloni-based system average 20% slower than the non-li system.

VI. Design Scaling using Circuit Example

We use a finite impulse response (FIR) cascade as a benchmark to test the efficiency of the LID systems as the chip is filled. A cascaded FIR module is a number of FIR modules connected together in a feedforward fashion. In our case each FIR module is the pearl inside of a LID wrapper. In between each wrapper we have latency-insensitive communication channels where we can insert pipeline elements. For the non-li system manual pipelining between FIR modules could be easily done. Since we are not using this design example as a comparison of design effort between LID and non-li design but instead as a high speed design example to compare design style efficiency it is appropriate to use it as a benchmark, as also was done in [4].

The FIR module consists of 13 chained DSPs. Each DSP in the Stratix 10 and Arria 10 FPGA contain two multiplies and an adder tree that connects the output to the chained input of the adjacent DSP. The filter has 51-tap symmetric coefficients; therefore, chain adders in the soft logic are used to implement a pre-adder for the inputs before going into the multiplier. The module's inputs and outputs are 17-bits wide. The module was designed to operate at at the fastest speed of the respective DSP block on the FPGAs used: this was 550 MHz for Arria 10 and 750 MHz for Stratix 10.

Fig. 9 shows operating frequency as the number of FIR modules increases to fill the chip. The plot shows the result for both no pipelining elements between FIR modules (dashed lines) and with the addition of pipelining elements (solid lines). Since Stratix 10 is a bigger device we tested the designs using four pipeline stages between FIR modules, while for Arria 10 we use two. For the Arria 10 design at zero pipeline stages the non-LI system does the best and the LI-systems cost some speed overhead. Beyond 20 modules we see a frequency drop; however, as we pipeline more we can handle larger designs. With two pipeline stages, designs with 40 FIR modules can gain a significant frequency improvement.

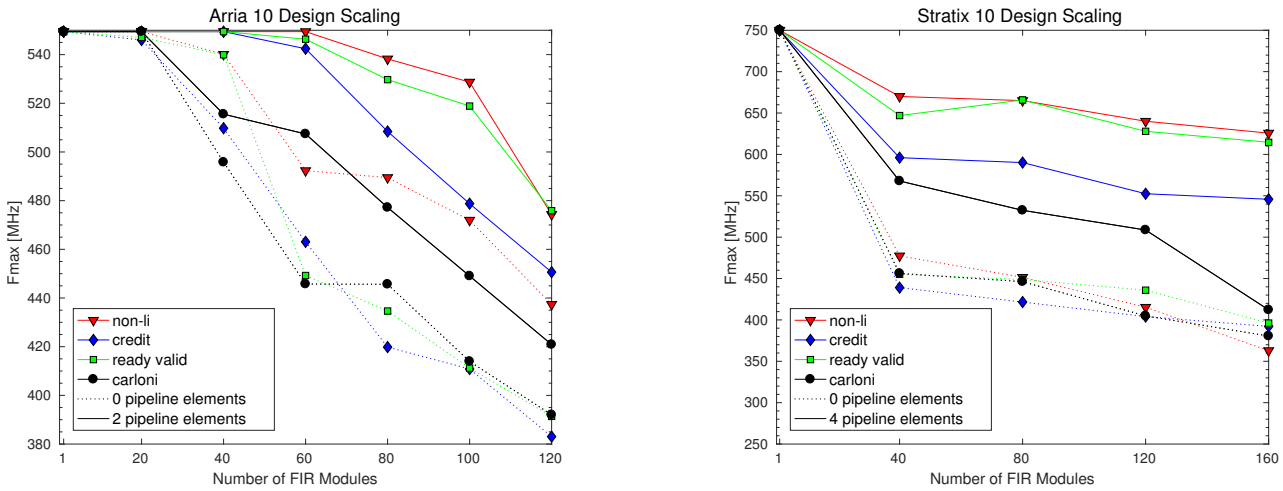

Fig. 9: Average operating frequency across five seeds as design size increases with and without system-level pipelining.

As the design size increases the frequency drops rapidly suggesting that more pipeline stages would be necessary to improve operating frequency. This highlights the importance of decoupling the design of the interconnect from the design of the IP-blocks so that it is easier to place pipeline elements on long connections.

In addition, at two pipeline stages in the Arria 10 design the ready-valid system performs nearly as well as the non-LI system. The credit-based system performs worse than the ready-valid system because of the additional logic needed for the credit counters. While Sec. V-A showed that the credit based system can run at high frequency, the counter logic slightly increases the size of the placement problem and appears to lead to less optimal placement in these larger designs. Finally the Carloni-based system has the lowest performance, matching the treads of Sec. V-A.

On the Stratix 10 design scaling plot in Fig. 9 we see a similar difference between the performance of the credit-based system and the ready-valid system. We also still see that the Carloni-based system again has lower performance than the the other solutions. The only difference verses Arria 10 is with zero pipeline stages all systems perform relatively similar. This may be due to better placement of BRAMs used for FIFOs.

Overall, the trends seen using the FIR benchmark are similar to those seen in Sec. V: ready-valid can achieve a similar speed to the non-li design and Carloni-based LID achieves the slowest speed. One result that is different is that the credit-based system while still better than Carloni-based now performs worse than the ready-valid system.

VII. Related Work

There are several different system level design approaches for FPGAs. In this section we compare and discuss recent work to improve system level design in FPGAs.

A system can be integrated at the HDL level but this is time consuming so many designers now use system integration tools like Quartus Platform Designer and Vivado IP Integrator [18], [19]. These tools help to abstract the design of the interconnect. However, unlike LID these tools require latency to be specified by the designer, leading to tedious exploration of latency options. For example, using the Altera Qsys system integration tool it took multiple CAD runs using the max tool effort to achieve a timing-closed placement for both small and large designs [20]. The academic tool in [21] is similar but supports a wider range of network topologies which can lead to area savings.

Some work to minimize the design effort for interconnect timing closure through automation has been done in [22]. It showed 43% area can be saved using a linear programing method to synchronize communication as opposed to FIFO based synchronization. However, this work assumes fixed latency, and only applies to streaming communication. Other work to ease the burden on developers focuses on fine-grained interconnect [23] and not on the global wiring delay problem.

Many architectural solutions have been proposed for solving the global wiring problem. One solution is through Globally Asynchronous Locally Synchronous Architectures (GALS) [24], [25]. This FPGA architecture independently clocks global and local interconnect. It assumes the use of asynchronous communication for global signals while maintaining synchronous communication locally. Unfortunately, this architecture requires radical clock network changes as well as major CAD tool changes.

Embedded Hard Networks-on-Chip (NoC) are another architectural solution [26], [27]. This approach hardens packet-switched routers within the FPGA and uses the resulting NoC for system level communication. The routers consist of arbiters, virtual channels, and credit-based FIFOs and are linked by high-speed dedicated wiring. They are introduced as a solution to global interconnect speed problems and are ideally used as a latency-insensitive system (through the use of lightweight LI wrappers with size less than 10 equivalent LABs). NoCs show significant gains including reducing routing utilization by 40% and improving frequency up to 80%. Thus, they are a promising candidate for future commercial FPGAs and can interface well with LID.

FPGA-optimized soft NoCs such as hoplite [28], CONNECT [29], and split-merge [30] also present a latency insensitive interface. They are efficient when modules' communication patterns vary over time; however, for fixed communication, the small LIDs we present are more efficient.

Other forms of LID have also been studied. Elastic systems are a form of LID that uses a different protocol than Carloni LID but use a similar form of pipelining elements as relay stations [31]. Another form of LID uses a clock schedule that determines the validity of the modules before design placement [32]. This approach allows for simple pipelining elements and can guarantee optimal throughput for the design. However, this approach does not allow for additional pipeline registers to be placed later in the CAD flow.

LID has also been previously used in FPGAs. The authors of [33] introduced Latency-insensitive Environment for Application Programming (LEAP) which is an overlay architecture to abstract FPGA hardware from the designer. LEAP leverages LI channels to communicate across the chip from multiple FPGAs to multiple CPUs, to provide a consistent interface to their services as well as physical interfaces, and to enable design portability. A multi-FPGA design is connected through asynchronous FIFOs at the chip boundaries, thereby abstracting the use of multiple FPGAs [34].

VIII. Conclusion

The increasing reliance on latency for global communication on FPGAs to achive timing closure motivates investigation of FPGA-friendly LID. As we have seen we require 60 stages of pipelining to communicate across a Stratix 10 chip at maximum frequency. This number will only increase with newer processes generations.

While the motivation for LID is clear, the best implementation on an FPGA is not. We compare three types of Latency insenstive design that can be used on FPGAs to ease the timing closure problem. The FIFO buffering solutions for LID (credit-based and ready-valid) are 2x more area efficient and 18% more speed efficient than the Carloni-based system on the traditional Aria 10 FPGA. Architectures with hyper-registers benefit even more. On Stratix 10, the FIFO buffering solutions for latency insensitive design are 3x more area efficient than Carloni-based ones. Compared to non-LI, LID systems only add a small area of less than 7 equivalent LABs to encapsulate the design modules for input port widths of 40-bits or less. FIFO buffering solutions add only a small (20%) area to pipeline elements compared to non-LI systems; this extra area arises from the need to pipeline a back-pressure signal.

The FIFO buffering solutions can be used at a small area overhead; however, with regards to operating speed we have found that one FIFO buffering solution is better than the other. The ready-valid design style performs the best. It is able to maintain little to no speed difference compared to the non-li system as design size scales, whereas the credit-based style degrades in frequency due to placement effects.

We have shown that with the right latency-insensitive style, LID can be efficiently used in FPGAs without significant area overhead and with little to no timing overhead. Ultimately to take full advantage of LID, automated pipelining tools should be built in to the CAD flow to increase designer productivity, so we consider this as important future work.

Acknowledgment

The authors would like to thank Mohamed Abdelfattah and Kevin Murray for providing useful insights for this work. This research was supported by the NSERC/Intel Industrial Research Chair in Programmable Silicon.

References

[1] Intel FPGA, "Intel Stratix 10 Product Table," 2018.
[2] Intel FPGA, "Intel Arria 10 Product Table," 2017.
[3] R. Ho *et al.*, "The Future of Wires," *Proceedings of the IEEE*, 2001.
[4] K. E. Murray and V. Betz, "Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs," in *FPGA*, 2014.
[5] K. E. Murray and V. Betz, "HETRIS: Adaptive Floorplanning for Heterogeneous FPGAs," in *FPT*, 2015.
[6] A. M. Caulfield *et al.*, "A Cloud-scale Acceleration Architecture," in *MICRO*, 2016.
[7] L. P. Carloni *et al.*, "A Methodology for Correct-by-construction Latency Insensitive Design," in *ICCAD*, 1999.
[8] L. P. Carloni *et al.*, "Theory of Latency-insensitive Design," *TCAD*, 2001.
[9] D. Lewis *et al.*, "The Stratix 10 Highly Pipelined FPGA Architecture," in *FPGA*, 2016.
[10] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with Latency in SoC Design," *MICRO*, 2002.
[11] L. P. Carloni, "From Latency-insensitive Design to Communication-based System-level Design," *Proceedings of the IEEE*, 2015.
[12] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency Insensitive Systems," in *DAC*, 2000.
[13] Intel, "Avalon Interface Specifications (MNL-AVABUSREF)," 2018.
[14] Xilinx, "AXI Interconnect Referance Guide (UG761)," 2011.
[15] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. 2004.
[16] R. Rashid *et al.*, "Comparing Performance, Productivity and Scalability of the TILT Overlay Processor to OpenCL HLS," in *FPT*, 2014.
[17] Intel FPGA, "Hyper-pipelining for Stratix 10 Designs (AN715)," 2015.
[18] Intel FPGA, "Intel Quartus Prime Handbook (QPS5V1)," 2018.
[19] Xilinx, "Vivado Design Suite User Guide (UG994)," 2018.
[20] M. S. Abdelfattah and V. Betz, "The Case for Embedded Networks on Chip on Field-programmable Gate Arrays," *MICRO*, 2014.
[21] A. Rodionov and J. Rose, "Automatic FPGA System and Interconnect Construction with Multicast and Customizable Topology," in *FPT*, 2015.
[22] A. Rodionov and J. Rose, "Synchronization Constraints for Interconnect Synthesis.," in *FPGA*, 2017.
[23] A. Rodionov *et al.*, "Fine-grained Interconnect Synthesis," *TRETS*, 2016.
[24] A. Royal and P. Y. Cheung, "Globally Asynchronous Locally Synchronous FPGA Architectures," in *FPL*, 2003.
[25] X. Jia and R. Vemuri, "The GAPLA: a Globally Asynchronous Locally Synchronous FPGA Architecture," in *FCCM*, 2005.
[26] M. S. Abdelfattah *et al.*, "Take the Highway: Design for Embedded NoCs on FPGAs," in *FPGA*, 2015.
[27] T. Liu, N. K. Dumpala, and R. Tessier, "Hybrid Hard NoCs for Efficient FPGA Communication," in *FPT*, 2016.
[28] N. Kapre and J. Gray, "Hoplite: Building Austere Overlay NoCs for FPGAs," in *FPL*, 2015.
[29] M. Papamichael and J. Hoe, "CONNECT: Re-examining Conventional Wisdom for Designing NOCs in the Context of FPGAs," in *FPGA*, 2012.
[30] Y. Huan and A. DeHon, "FPGA Optimized Packet-switched NoC Using Split and Merge Primitives," in *FPT*, 2012.
[31] J. Cortadella *et al.*, "Synthesis of Synchronous Elastic Architectures," in *DAC*, 2006.
[32] M. R. Casu and L. Macchiarulo, "A New Approach to Latency Insensitive Design," in *DAC*, 2004.
[33] K. Fleming and M. Adler, "The LEAP FPGA Operating System," in *FPGAs for Software Programmers*, 2016.
[34] K. E. Fleming *et al.*, "Leveraging Latency-insensitivity to Ease Multiple FPGA Design," in *FPGA*, 2012.