System-Level Retiming and Pipelining

Girish Venkataramani
MathWorks, Inc.
Natick, MA 01760, USA
gvenkata@mathworks.com

Yongfeng Gu
MathWorks, Inc.
Natick, MA 01760
ygu@mathworks.com

Abstract—In this paper, we examine retiming and pipelining in the context of system-level optimization techniques. Our main contributions are: (a) functionally equivalent retiming and delay balancing as necessary techniques for pipelining and retiming system-level graphs while maintaining numerical fidelity, and (b) clock-rate pipelining, as a new technique that leverages the knowledge of multi-rate design spec to pipeline multi-cycle paths. All these techniques have been implemented within HDL Coder®, a tool that generates synthesizable HDL code from Simulink® and MATLAB®.

I. INTRODUCTION

FPGAs are finding their way into applications in several domains, such as signal processing, image/video processing, communications, automotive, aerospace, medical devices and radar. This widespread use along with increasing design complexity has resulted in the raising of the level of abstraction for FPGA design. While traditionally FPGAs were programmed in hand-written RTL (VHDL or Verilog), the past decade has seen the advent of many FPGA design tools like HDL Coder™ [18] that generate synthesizable RTL from Simulink® [20] and MATLAB® [19]. Similarly, various tools can generate RTL from System-C and other C-like languages [12], [8], [6], [5], [30], [9]. This is because controls engineers, signal processing engineers and image processing engineers want to leverage the power of FPGAs in their new state-of-the-art platform but most of these system-level engineers have little knowledge of FPGAs or RTL programming.

There is consensus amongst the research community that introducing design optimizations, like pipelining, scheduling and resource sharing, early in the design flow has the potential to: (a) make much deeper impact on the final quality of results (QoR), (b) find bugs more quickly making verification easier and (c) overall, greatly help in shrinking the design cycle time. In this context, we are particularly interested in timing optimizations that can be applied at the system-level. On the one hand, high-level synthesis tools traditionally employ scheduling [11], [12], [22] as the primary timing optimization technique that can freely introduce new pipeline registers and may even alter the timing of the circuit. On the other hand, RTL synthesis tools are bound by the timing contract expressed in the RTL architecture. While they cannot introduce new pipelines, alter timing or introduce new clocks, they can apply optimizations like retiming [13] and buffer minimization [14] that respect the timing contract but improve the timing of the circuit.

In this paper, we propose leveraging the techniques from both high-level synthesis tools as well as RTL synthesis tools.

Our main contributions are new system-level timing optimizations that capitalize on the high-level language semantics. We propose the following main ideas:

1) System-level, functionally-equivalent retiming: The traditional Leiserson/Saxe retiming algorithm [13] was formulated for an RTL graph comprising of logic and flip-flops or registers. When applying it in the system-level context, one must solve the initial state computation problem for functional equivalence [16], [29], [28], [24]. This is hard because, logic at the system-level may be in the form of adders, multipliers or even FFT modules instead of AND, OR, NOT gates in logic synthesis. Instead of computing initial states, we define the concept of retiming safety under which initial state computation is not needed.

2) Pipelining and delay balancing: Often system designers know more about their applications than can be inferred either in high-level or RTL synthesis tools, e.g., they may know that a pipeline delay is required after a particular algorithmic module. To support such a workflow, we propose delay balancing as a novel algorithm to balance new pipeline delays across arbitrary dataflow graphs as well as across multi-rate designs.

3) Multi-rate pipelining: A common modeling paradigm at the system-level is multi-rate systems. This is an abstraction that is known at the system-level but is ambiguous and practically unknown at the RTL level. We describe two techniques to achieve high performance in these cases: (a) automatic constraint generation and (b) clock-rate pipelining. The former automatically generates multi-cycle path constraints that is passed on to the downstream synthesis tool. The latter, pipelines and retimes through multi-cycle paths without introducing additional latency.

We describe all these techniques within HDL Coder™ [18], a system-design tool that automatically generates synthesizable RTL from Simulink [20] and MATLAB [19] and also allows the user to directly program FPGA boards. In Section III, we describe our compiler’s framework and setup. Section IV motivates the need for ad-hoc pipelining and delay balancing. Section V describes the concept of retiming safety for provably functionally equivalent retiming. Section VI describes pipelining and retiming in multi-rate designs. In Section VII, we show the effectiveness of these techniques on real-world applications. We conclude in Section VIII.
II. Related Work

Retiming was first introduced by Leiserson and Saxe [13] as a graph transformation to improve critical path timing. Several researchers have built upon this to use retiming to optimize different aspects of the circuit, e.g., min-area [15], [3], sequential synthesis [17], low-power [23].

Timing optimization at the system-level has been traditionally addressed through scheduling methods in high-level synthesis [12], [22]. Through scheduling, the design’s timing may be improved by introducing pipeline registers between long chains of computation. Several enhancements and improvements to the scheduling algorithm have been proposed, e.g., force-based scheduling [25], modulo scheduling [27], etc. While the retiming algorithm and its variants concern itself with synchronous circuits, several techniques such as pipeline balancing, slack matching [26], [4], [31] and buffer sizing [14] have been proposed for latency-insensitive circuits and asynchronous circuit design.

This paper describes new system-level timing optimization techniques. With regards to retiming, one of the fundamental issues is finding equivalent initial states of retimed registers so that functional equivalence is preserved. This is an NP-hard problem and several researchers have proposed novel techniques to compute the initial state [16], [29], [28], [24]. Instead of computing initial states, we formally prove that there exists a set of properties and constraints on a retiming graph, which when satisfied implies that initial state computation is not necessary for retimed registers. Thus, we define retiming safety for a graph and then apply the traditional retiming algorithm [13] without having to fix initial states.

We also describe the importance of ad-hoc pipelining for system designers. However, arbitrarily adding pipeline registers in the design requires supporting algorithms that can perform delay balancing across parallel paths and also pipeline through multi-cycle register paths. To our knowledge we are the first to propose system-level techniques to optimize timing of multirate systems.

III. HDL Coder: Simulink to HDL

We begin by describing HDL Coder as shown in Fig. 1. The source language is Simulink [20], a graphical system-level language. HDL Coder converts Simulink to an intermediate representation called PIR (Section III-A), performs the proposed timing optimizations on this IR, and the back-end generates equivalent synthesizable RTL in either VHDL or Verilog. HDL Coder prescribes to the model-based hardware design flow [32] where design, implementation, testing and verification are a continuous part of the design flow.

Simulink is a rich graphical specification language used to describe algorithms and applications in signal processing, communications, image processing and controls design. The user draws the algorithm as a graph on the Simulink canvas, where each node or block has a well-defined functionality. Blocks could range from a simple adder or multiplier to complex transforms like an FFT [20]. The blocks can also be user-defined, wherein the algorithms are written in MATLAB [19] or Stateflow® [21], a state-machine modeling language. Edges in the Simulink graph are called signals, which connect the output of one block to the input of another. Simulink signals have well-defined properties, like data-type and a sample time. The sample time specifies the rate at which data on that signal is sampled. This is has important implications to our system design tool because an explicit notion of time is modeled in Simulink allowing HDL Coder to make smart decisions during timing optimizations.

A. Parallel Intermediate Representation (PIR)

The front-end of HDL Coder translates the Simulink model to an intermediate representation (IR) called Parallel Intermediate Representation (PIR), formally defined as $PIR = (V,E)$, where $v \in V$ and $e \in E$ are the nodes and edges of the PIR graph respectively. $PI$ and $PO$ represent the primary inputs and outputs of the graph respectively. Every edge, $e \in E$, has exactly one driving node, referred to as $Src(e) \in V$ and may fanout to a number of destination nodes, $Dst(e) \subseteq V$. The inputs of a node, $v \in V$, are specified in terms of its input edges, $inedges(v) \subseteq E$ or its input nodes, $ins(v) \subseteq V$, where $ins(v) = \cup_{e \in inedges(v)} Src(e)$.

Of particular importance to timing analysis is the fact that $time$ is a first-order abstraction in Simulink. In the fixed-step, discrete solver mode [20] of Simulink, the model behaves very similar to a synchronous RTL circuit with respect to timing. The edges of PIR are associated with a sample time that specifies the frequency, in seconds, at which data on the edge is sampled by the consumers of the edge; this is represented as $t(e) \in T$, where $T$ is a set of all discrete rates represented in the PIR graph.

Further, there are two types of PIR nodes of particular interest since they operate on the Simulink timing abstraction:

- Delay: A delay node, represented as $Z^{-1}$, maps to a register in the RTL. The semantics of this block is that the data at its input is available in the next Simulink time step at the given rate of the delay’s output signal. By extension, $Z^{-k}$, maps to a shift-register or line-buffer of length $k$.
- Rate-transition: which changes the sampling rate of a signal path. It can either be an upsampling or a downsampling rate-transition. If the input sampling rate of the component is $T$ seconds and the rate-transition factor is $K$, then the sampling rate of the output of
an upsampling rate-transition is \( \frac{K}{N} \) seconds, while the sampling rate of the output of a downsampling rate-transition is \( T \times K \) seconds.

All rate transitions occur in integer multiples of the input rate. Thus, for the whole PIR graph, there exists a discrete set of sampling rates, represented by \( T \), which includes a base sampling rate, refered to as \( BSR = GCD(\forall t \in T) \). The PIR graph is the input to the retiming and pipelining techniques described in this paper.

### IV. Pipelining and Delay Balancing

When creating the algorithm in Simulink, the systems engineer seldom thinks about pipelining and timing of the final RTL circuit. On the contrary, it is better to focus on the algorithm rather than implementation details of pipelining. Once algorithmic correctness is established, the user then focuses on implementation efficiency. There are two common for the engineer to want to apply ad-hoc pipelining — first, pipelining long latency operations and, second, manual ad-hoc pipelining.

#### A. Pipelining Necessities

The source Simulink model may contain complicated blocks like division, square-root, trigonometric function etc. Since Simulink is timed, the algorithm specifies this block’s operation at a particular sample-rate. Let us consider an example math equation, \( y = x + \sqrt{u} \) and the sample-time of \( y \) is 20ns. In Simulink, all math operations are combinatorial — in other words, outputs are computed in the same step the input is presented. This means that the RTL must also perform a square-root operation within a single clock cycle, if the RTL is to be deemed cycle-accurate with respect to the timing of the source Simulink model.

Clearly, a combinatorial square-root operation in RTL is impractical. Thus, HDL Coder provides a number of pipelined implementations of square-root, e.g., Newton-Rhapson approximation. This will result in a pipelined square-root implementation of (say) 4 cycles. Pipelining is not restricted to just long-latency operations. The designer may know something of the circuit’s behavior and choose to arbitrarily insert a pipeline anywhere in the design graph. This can greatly improve the circuit timing.

#### B. Delay Balancing

When pipeline delays are introduced in the model for performance reasons, the tool must ensure numerical equivalence. All delays must be balanced by inserting matching delays on parallel paths. Further, multi-rate designs pose additional concerns because samples are discarded when downsampling from a fast rate to a slower rate. The delay balancing problem is solved by defining a delay transfer function through the PIR graph.

For a node, \( v \in V \), let the number of pipeline delays introduced at \( v \) be represented by \( \text{PipeDelay}(v) = (n, r) \), where \( n \) delays are introduced at a sampling rate of \( r = t(v) \) seconds. For example, if the Newton-Rhapson implementation of the square-root node, say \( v_{\text{sqrt}} \), introduces 4 cycles of delay, then \( \text{PipeDelay}(v_{\text{sqrt}}) = (4, r) \), where \( r = t(v_{\text{sqrt}}) \).

The above equations formally define a propagation algorithm to find merge points in the PIR graph and insert matching delays, \( \text{Deficit}(e) \), on the other incoming edges at the merge point. To compute the deficit, we normalize the delay numbers along the incoming paths at each merge point using the base sampling rate of PIR, \( BSR \). When performing any math on a pipeline delay, the normalized delays are used. The term, \( \text{InDelay}(v) \), represents the maximum new delay introduced due to pipeline delays along any path from the primary inputs to node \( v \). If the delays along the incoming edges of the merge point are unbalanced, then matching delays are inserted, i.e., \( \text{Deficit}(e) \).

Besides merge points created by dataflow paths, delays must also be balanced at rate-transition boundaries, specifically, downsampling rate-transitions. Consider Fig. 2, which shows a rate-transition node, say \( v \) that downsamples by a factor of 4x; thus if its input sample time is \( t \), then its output sample time is \( 4t \). The operation of this node is to discard 3 samples and send the 4th sample into output (row 2 of Fig. 2). However, if \( \text{InDelay}(v) = 1 \), then the samples are phase-shifted by one cycle (row 3 of Fig. 2), in which case the downsample block would pick the wrong sample. Delay balancing fixes this problem by re-aligning the phase of the downsample by adding sufficient delay on the input of the downsample so as to align to the next correct phase of operation. Specifically:

![Fig. 2. The timing of a downsample-by-4 node. The table shows the input value, \( u_1 \) and that the output, \( y_1 \), picks up every 4th cycle input. If a 1-cycle pipeline delay (\( \text{PipeDelay} = 1 \)) is introduced upstream of the downsample, then its inputs are phase shifted by a cycle as shown in the 3rd row. In this case, delay balancing aligns it to the right phase by inserting 3 more delays as shown in the 4th row.](image_url)
the initial value of delay is 0. However, if the delay is moved backward across a bitwise-not node, the output sequence becomes 0, breaking functional equivalence.

If \( DsFactor(v) \) is the downsampling factor, then the phase offset is given by \( Offset(v) \). We insert \( Deficit(v) \) delays on the inputs of \( v \) to align to the next correct phase of the downsampling, as shown in row 4 of Fig. 2. Now, we are picking the right sample but it is simply delayed by one slow cycle or 4 fast cycles.

V. FUNCTIONALLY EQUIVALENT RETIMING

After pipelining the design by inserting ad-hoc pipelines and delay balancing, we apply a modified version of the Leiserson/Saxe retiming algorithm [13]. We show in this section that if we were to simply apply the unmodified retiming algorithm on the system (PIR) graph, it may break functional equivalence yielding a circuit producing wrong answers. For illustration, we explain this with a simple example in Fig. 3. If in the original design, the delay node is at the output of the bitwise- NOT node, then output of the circuit is \( \{0, 0\} \) because the initial (reset) value of delay is 0. However, if the delay is retimed to the input of the NOT node, then the output changes to \( \{1, 0\} \), thus violating functional equivalence in this trivial case. Clearly, in the presence of more complex arithmetic and feedback loops, the situation becomes more dire and probably even irrecoverable. Our main contribution in this section is to propose a set of formal conditions, which if satisfied by the graph, will guarantee functional equivalence no matter how the graph is retimed. Based on these conditions, we propose a modified application of the retiming algorithm on the PIR graph without violating functional equivalence.

The primary insight is the fact that when a register is retimed, its initial value may have to also be changed to maintain functional equivalence. For example, in Fig. 3, if we changed the delay’s initial value to 1, then we would have produced \( \{0, 0\} \) and maintained functional equivalence. Thus the challenge when retiming is calculating initial values to provide an equivalent initial state for retimed delays; this is NP-hard for arbitrary logic. In some cases, finding an equivalent initial state is impossible unless additional logic is added, as shown in Fig. 4. Here, the two delays have different initial values; thus, if we retime them across the NOT node, it is impossible to set a fixed initial value.

The common solution for RTL synthesis tools is to reduce the complexity of logic to be analyzed. For example, Synopsys SynplifyPro [1], does not move registers across sequential components, instantiated components, carry and cascade chains, etc. Others have devised ways to determine the new initial condition of the retimed registers [16], [29], [28], [24]. We take a different approach. Instead of determining the initial values for retimed registers, we define conditions under which the initial values of retimed registers do not have to change and the circuit is still functionally equivalent. Based on this, we propose a novel algorithm called functional equivalent retiming or FE retiming. We begin by formally defining the conditions for retiming safety.

**Theorem 1:** For a node, \( v \in V \), let \( y(t) = F(x(t), s(t)) \) be its output function at time \( t \), and \( s(t+1) = S(x(t), s(t)) \), be its state update function at time \( t \); \( x(t) \) and \( s(t) \) are its input and internal state variables at time \( t \). For a given constant \( m \), \( v \) is said to be retiming-safe in \( m \) and a register can safely be moved from output to input (or vice-versa) iff it satisfies the following two conditions:

1. \( F(m, s(0)) = m \).
2. \( S(m, s(0)) = s(0) \).

**Proof:** The conditions simply check that the initial values of \( v \) do not change when the input is \( m \) and that the output is also always \( m \). Consider Fig. 5. When the delay is on the node’s output, the graph’s output is \( \{m, y(t)\} \). When the delay is retimed to the input of \( v \), the input to \( v \) changes from \( \{x(t)\} \) to \( \{m, x(t-1)\} \). Because \( S(m, s(0)) = s(0) \), the state variable does not change for the first cycle, \( v \) remaining in its initial state. In the second cycle, because the input to \( v \) resumes the original input sequence and \( v \) itself is still in the initial state, \( v \) behaves as same as in the pre-retimed sub-graph only with one cycle delay. Therefore the output sequence of \( v \) is \( \{y(t)\} = \{m, y(t-1)\} \). This is also the output sequence of the retimed sub-graph. Moving delays forward across \( v \) just reverses these steps and has the same result.

The intuition behind the conditions is to define the steady-state of the circuit during retiming. For a given input value, \( m \), if the circuit produces the same output value, \( m \), and it does not change the internal state, then we say that it is retiming-safe in
Based on this, we can develop the notion of a retiming-safe sub-graph.

**Lemma 1:** If nodes $u$ and $v$ are connected back to back and both are retiming-safe in $m$, then the path through $u$ and $v$ is also retiming-safe in $m$.

**Proof:** Assume that $u$ is at the input of $v$. Since they are both retiming-safe in $m$, both satisfy $F(m,s(0)) = m$. When $u$ takes input $m$, it outputs $m$ on the edge connecting the nodes; thus input to $v$ is also $m$ and thus it also outputs $m$. Therefore as an entity, the path of $u$ followed by $v$ satisfies $F(m,s(0)) = m$. By the same logic, $S(m,s(0)) = s(0)$ for both nodes and thus for the path as a whole. Thus, the whole path is retiming-safe in $m$.

**Theorem 2:** Moving a delay with initial value of $m$ in a graph does not change the graph’s initial state, if all nodes of the graph are retiming-safe in $m$, and thus the graph is retiming-safe in $m$.

**Proof:** If all nodes in the graph are retiming-safe in $m$, and $m$ is injected in to the graph’s inputs, then all edges in the graph acquire a value of $m$. Thus, if a delay, whose initial value is also $m$, is moved arbitrarily across a node, it does not change the value on any edge in the graph and thus does not affect any of the initial state.

Thus, if we can construct a retiming-safe (in $m$) graph where all nodes are retiming-safe in $m$, then we can safely apply the traditional retiming algorithm [13] without having to re-compute initial state of retimed registers. Practically, we use $m = 0$. This is because most registers have an initial state of 0 and most arithmetic operations are retiming-safe when $m = 0$, e.g., for addition and multiplication, $F(0) = 0$ irrespective of their other properties.

Our FE retiming algorithm has two phases: (1) partition PIR to identify retiming-safe sub-graphs; (2) apply the traditional retiming algorithm on these sub-graphs. The following algorithm finds all retiming sub-graphs in $PIR = (V,E)$ with a time complexity of $O(|V| + |E|)$:

1. **function** Partition($PIR = (V,E)$)
2. **while** not all node in $V$ not visited
3. **do**
4. Find a node $sv$ in $V$ that has not been visited
5. Create an empty node set $V_{rt}$
6. Relax($sv,V_{rt}$)
7. **if** $V_{rt}$ is not empty **then**
8. With $V_{rt}$, create a sub-graph $G_{rt} = < V_{rt}, E_{rt} >$, where $E_{rt}$ contains all edges in $E$ that connect nodes in $V_{rt}$
9. **end if**
10. **end while**
11. **end function**

In the second phase, traditional retiming is applied on all retiming sub-graphs found in the first phase. The FE retiming algorithm has several advantages: (a) it is fast, (b) the expensive retiming phase is performed on smaller sub-graphs, and (c) it does not need to compute equivalent initial state, (d) it does not require additional logic to compute initial state, and (e) it is capable of handling complex nodes. Since PIR has a finite set of node-types, we define retiming-safety for each node type such making it easy to define retiming-safety for the graph. For user-defined functions, e.g., using MATLAB functions or Stateflow charts, we use constant-propagation to check if the retiming-safety conditions are met.

**VI. MULTI-RATE SYSTEM PIPELINING**

Optimizing the design at the system-level means that we can leverage abstractions and knowledge that is available at the system-level but are lowered or are ambiguous at the RTL level. One such property is sampling rate relationships. It is quite common to model applications in the signal processing and communications domain as multi-rate systems, e.g., decimators and interpolators, upsampling and downsampling are common signal processing operations. In domains like control systems, the sensor data from the plant model is typically streamed to the FPGA controller at the rate of KHz, but the controller itself is running at the MHz frequency.
Multi-rate abstraction is clearly a system-level concept as modeled in Simulink. There are actually two choices to implement multi-rate in RTL: first, use multiple clocks to model a clock for each rate domain, and second, use a single clock and clock-enables to control when registers are enabled at different rates. The former is a design choice and may often be undesirable because of synchronization requirements at clock crossing boundaries. Further, if the number of rates in the design is too high, then it may be inefficient to model a clock per rate, considering that the Digital Clock Manager (DCM) on most FPGAs can only support a finite number of clocks. Thus the latter, single-clock with clock enables, or even multiple-clocks (for a subset of rates) with clock-enables (for the other rates) is a common design paradigm. HDL Coder supports both modes. For the purposes of this discussion, we focus on the more general case, where rates are implemented as clock-enables.

When rates are implemented as clock enables, as shown in Fig. 6, we already lose the high-level abstraction of mult rates. For example, the Simulink signals have an explicitly defined sampling rate as shown. However, a common RTL implementation may use a timing controller to produce clock-enables that are generating pulses at different rates. For example, the signal en8 is generating a pulse every 8 clock cycles whereas the signal en2 is generating a pulse every 2 clock cycles, meaning data can take 2 cycles to propagate between the registers, but this is very difficult for RTL tools to infer. In this section, we show two system-level techniques to address this problem: multi-cycle constraint generation and clock-rate pipelining.

A. Multi-cycle constraint generation

To address the multi-rate problem, most synthesis tools including Xilinx ISE, Vivado and Altera Quartus, allow for multi-cycle constraints specification. For our example in Fig. 6, we could specify a multi-cycle constraint in the .ucf format (used by Xilinx ISE) [7]:

```text
TIMESPEC "TS_multi_path" = FROM "dFast" TO "dSlow" TS_clk*2
```

Our compiler knows of this timing relationship and can automatically generate these constraints during RTL code generation. In PIR, we find every register-to-register path. For a given register-to-register path, with source register, \( n_{src} \), and destination register, \( n_{dst} \), we define the path multiplier value as \( \text{Min}(\text{Scale}(n_{src}), \text{Scale}(n_{dst})) \), where \( \text{Scale}(v) \), is rate ratio of \( t(v) \) to the base sampling rate, \( BSR \). For HDL code-generation, our compiler restricts the source Simulink model to only perform perfect rate-transitions. In other words, data can change only as quickly as the faster rate between the registers. Once the path multiplier value is determined, the multi-cycle constraint can be generated as below:

\[
\text{pm} = \text{Min}(\text{Scale}(n_{src}, n_{dst}))
\]

```text
TIMESPEC "TS_multi_path" = FROM n_src TO n_dst TS_clk*pm
```

B. Clock-rate Pipelining

Generating multi-cycle path constraints essentially provides hints to the RTL synthesis tool about the timing of various register paths in the design. We can, however, do more from a system-level perspective. We can improve the timing of the circuit using a technique we call clock-rate pipelining. This technique leverages multi-cycle paths and inserts registers that run at the clock-rate rather than at the functional rate. Consider the example path from register, \( src \) to register \( dst \). Assume this is a slow path running \( N \) times slower than the clock rate. This can be modeled in Simulink simply by setting the right sample time on the signals. This means there are \( N \) clock cycles available to complete the computation along this path, and \( (N-1) \) clock-rate pipelines can be inserted along this path to improve timing.

The real advantage of clock-rate pipelining is when optimizing feedback loops, which are known to be hard to pipeline. Consider a high-latency operation, e.g., floating point multiplication within a feedback loop. Typical floating-point library implementations [33], [2] define multi-cycle (on the order of 10 cycles) latencies. However, since Simulink is a timed system, the feedback loop processes a new sample every time step of the sampling rate at which the loop is operating. However, if we know this sampling rate, \( t(\text{loopreg}) \), is slower than the base sampling rate, \( BSR \), then the loop itself is a multi-cycle path, which means we can introduce as many as \( \text{Scale}(t(\text{loopreg})) - 1 \) delays in this loop, thus being able to pipeline such synchronous feedback loops without affecting numerical equivalence.

Clock-rate pipelining a general PIR graph is performed as follows. We first apply a partitioning algorithm to find maximal, combinational sub-graphs within PIR. If the sub-graph is operating at a rate slower than the clock-rate, then it is a candidate for clock-rate pipelining. A maximal combinational sub-graph, \( G_s = (V_s \subseteq V, E_s \subseteq E) \), is defined by the following properties:

- All primary outputs, \( PO_s \), of \( G_s \) are registers operating at the same sampling rate, i.e., \( t(n_1) = t(n_2) \), if \( n_1, n_2 \in PO_s \).
- All other nodes, i.e., \( (V_s - PO_s) \), are combinatorial nodes, i.e., have no state.
- The sampling rate of the sub-graph is given by \( t(G_s) = t(po) \), where \( po \in PO_s \).

Clock-rate pipelining is applied on such a sub-graph as long as it is operating at a rate slower than the clock-rate. Further, it implies that all paths in this sub-graph from primary inputs to primary outputs are multi-cycle paths, whose path multiplier is guaranteed to be \( \text{Scale}(t(G_s)) \). Clock-rate pipelining now involves inserting \( \text{Scale}(t(G_s)) - 1 \) new registers on every primary output of this sub-graph and then invoking the FE retiming algorithm described in Section V. The algorithmic details of clock-rate pipelining are summarized here:

1. Insert \( \text{Scale}(t(G_s)) - 1 \) new register nodes at the input of each primary output node.
2. Insert a rate-change node at the input of each primary input node. The rate change node converts from \( t(G_s) \) to the design’s base sampling rate, \( BSR \).
to 74 MHz using HDL Coder generated RTL. The results are encouraging — using pipelining, delay balancing and FE retiming the clock frequency can be boosted from 23 MHz to 74 MHz using HDL Coder generated RTL.

Building the Simulink model for functional correctness is the responsibility of the algorithm specialist. Using Simulink, the engineer can test the effectiveness of the algorithm by using image/video viewers and other visualization tools provided by MATLAB and Simulink. This can be an iterative process until the quality of the algorithm is satisfactory.

While the algorithm is being tuned in this manner, the algorithm specialist does not and need not insert implementation details like pipelining. Once the algorithm is stabilized, then HDL Coder is used to generate RTL for this design. This is the point at which implementation details like pipelining, delay balancing and retiming is added. The important thing is that the pipelining details do not pollute the original source model, which is still the algorithmic reference specification.

We synthesized the code generated for the corner detection model using Xilinx ISE and targeted a Virtex4 chip. The synthesis options were set aggressively with retiming option turned on. The clock frequency estimated post place and route is 23 MHz. We then introduced pipelining in the Gaussian filter and HDL Coder automatically runs delay balancing and FE retiming to generate pipelined RTL, which when synthesized for the same target, yielded a clock frequency of 74 MHz with little or no extra design effort. There are two primary reasons why synthesis tools could not do better: (a) they are bound by a timing contract and cannot introduce new pipelines, (b) even if pipeline registers were available, it is difficult for synthesis tools to move registers across complex algorithmic logic and module hierarchies due to the initial value problem.

### B. Motor Control: Clock-rate pipelining

To demonstrate the need for clock-rate pipelining, we turn to a control system application, a field-oriented controller. A common constraint in control system applications is that there is the plant and a digital controller, as shown in Fig. 8. The plant, which is the analog component of the design is generating sensor data every few microseconds, which corresponds to a few KHz sampling frequency. The controller, however, is running at full FPGA clock speed which is a few MHz. A sample model of a field-oriented controller is also shown in Fig. 8 as an example of such a modeling paradigm in Simulink.

We examined the design of a field oriented controller which calibrates the motor to run at a pre-specified speed and torque. The controller uses non-trivial math, performs discrete-time integration and also computes trigonometric functions in the speed controller component. The good news is that the controller is a slow path. The plant is generating samples at 25 KHz and timing requirements of the controller is that it must operate at 100 MHz without introducing any latency.

When RTL is synthesized without clock-rate pipelining, i.e., knowledge of multi-cycle paths, pipelining is inserted at the slow KHz rate. When synthesized to a Virtex6 FPGA using Xilinx ISE, the estimated clock frequency is 55 MHz. However, this could only be achieved by inserting five slow pipeline delays which operate at the 25 KHz rate. With clock-rate pipelining, we realize that there is a budget of about 4000 cycles in every slow rate path (ratio between 25 KHz and 100 MHz). Using this knowledge, FE retiming places the pipelines in strategic points to yield a synthesized design that operates.
Fig. 8. In motor control applications, the plant is generating samples every few microseconds, i.e., KHz sampling frequency, while the controller is running at FPGA clock frequency, i.e., MHz rate. For simulation speed, it is desirable to model the controller at KHz rate, but pipelining must be applied at the clock-rate. This is done by clock-rate pipelining.

at 166 MHz. Further, no new slow delays need to be inserted with clock-rate pipelining. This is about a 3x speedup without any new delays introduced.

VIII. CONCLUSIONS

We have proposed a number of new system-level timing optimization techniques. First, we have adapted and reformulated the traditional retiming algorithm and introduced retiming safety to address the initial value computation problem. Second, we have introduced the concept of ad-hoc pipelining to help hardware designers to arbitrarily insert pipeline registers at any location in the circuit and have shown how a technique called delay balancing can be used to balance all parallel path of the design. Third, we have shown how system tools can optimize timing in multi-rate designs by either automatically generating multi-cycle path constraints for RTL synthesis tools or by employing clock-rate pipelining. We have shown a 3x boost in clock frequency using these techniques in applications from image processing to motor control. All these techniques are automatically applied within HDL Coder, a system design tool that generates synthesizable RTL from Simulink and MATLAB models.

REFERENCES