# A Safe and Complete Gate-Level Register Retiming Algorithm

Babette van Antwerpen, Mike Hutton, Gregg Baeckler and Richard Yuan

Altera Corporation, 101 Innovation Drive, San Jose, CA 95134 (bantwerp,mhutton@altera.com)

# ABSTRACT

Retiming a netlist is an often-studied problem that is not often implemented in practice. Retiming is a relatively dangerous operation in the synthesis flow due to its effects on simulation, verification, debugging, issues such as meta-stability, and because the timing visibility early in the CAD flow is significantly less than would be desired. Many published algorithms also ignore important issues such as register compatibility due to secondary signals, don't touch constraints, common FPGA hardware such as RAM and carry chains and various illegal forms of register moves.

In this paper we will discuss our solution to the retiming problem which takes all these issues into account, provide empirical evidence of the viability of retiming on large industrial netlists, discuss the expected gains from retiming, and introduce better ways of analyzing and quantifying these gains. Despite the many additional constraints added to the problem, we show a 5.0% mean improvement in performance with retiming vs. without on a large set of industrial designs, and further that implementing retiming without attention to legality overstates the gain by half.

#### Keywords

Programmable logic, FPGA, synthesis, register retiming.

#### **1. INTRODUCTION**

Register retiming is a synthesis operation in which registers are moved across combinational gates in the netlist in order to balance the delay on paths in the netlist, and hence minimize the length of the longest path.

There are many solutions to retiming, but it is not often discussed in the context of production quality tools for various reasons: compile time, practical issues, legality problems and difficulties with simulation and verification. Nonetheless, there are significant gains to be had from a proper and safe implementation of register retiming. We find that some designs see huge gains in performance.

This paper outlines a complete algorithm for retiming in FPGA-based designs, in which we do not just solve the algorithmic problem, but do so efficiently, and with all of the pitfalls of retiming properly addressed. Though some papers have attempted to extend the pure algorithmic problem to some additional problems such as register power-up-conditions or reset signals, no paper that we are aware of has discussed and solved all of these and other issues that we will describe, or attempted to retime designs which are representative of the type of complicated logic found in industrial FPGA designs.

In Section 2 we introduce the classical retiming problem, some of the known algorithmic solutions, and some of the often ignored drawbacks of retiming. In Section 3 we give our algorithm for gate-level retiming, which is based on ideas from other papers as well as new approaches, and discuss how to deal with the practical issues required in any realistic CAD flow. Section 4 further discusses legality issues such as meta-stability, and shows the effects that implementing these restrictions will have on the quality of results. Section 5 gives our empirical results on a number of industrial benchmark designs, and we conclude in Section 6.

# 2. REGISTER RETIMING

The measure of performance in a synchronous netlist is the longest delay of any register-to-register path in ns (called r2r) or, more commonly, 1000/r2r which gives the maximum clock-speed in MHz at which the design can be implemented without forcing functional failure. We will call this measure fmax.

In a typical netlist, this worst case delay is not realized by all register-to-register paths, so an attractive option is to move registers across combinational gates in order to balance the delays among all paths, and hence minimize the worst-case delay.

A simplified diagram of a retiming operation is shown in Figures 1 and 2. Figure 1 shows a graph in which the worstcase register-to-register depth is three, with a pin-to-register



Figure 1: A netlist with logic depth three.



Figure 2: The netlist of Figure 1 after retiming. Logic depth is reduced to one.

and a register-to-pin depth of zero (clock signals are not drawn, all registers are assumed to have the same clock). Depths are measured in gates. By register retiming, we can balance the lengths of the combinational paths and make the input-to-register, register-to-register and register-to-output depths all equal to one. Figure 2 shows the result of this. Register H from Figure 1 is moved backwards over gate C, resulting in two new registers H1 and H2. Register E is duplicated: the original is feeding A and B, the duplicate is feeding gate D. The duplicate of E and register G are moved forward over gate D, resulting in register G1.

The retiming problem can be modeled as follows. Given is a directed graph G=(V, E). Each node v has a non-negative node delay d(v), and each edge e has a non-negative integer weight w(e), which represents the number of registers on this edge. A retiming of a graph is a function r that maps each node to an integer r(v). The retimed weight w<sub>r</sub>(e) of an edge e=(u,v) is w(e)+r(v)-r(u). A retiming is legal if the retimed weights of all edges are non-negative. The delay of a node in the graph G with retiming r is defined to be D<sub>r</sub>(v) = d(v) + max{ D<sub>r</sub>(u) | (u,v) in E and w<sub>r</sub>(u,v) = 0 }. The graph delay or clock period is the maximum of D<sub>r</sub>(v) over all nodes. Given a graph G=(V,E) with weights and node delays, the objective of the retiming problem is to find a retiming r such that the clock period is minimum.

The first algorithmic solution to the problem was given by Leiserson and Saxe [11]. They present a dynamic programming algorithm based on Bellman-Ford that finds a retiming for a given target clock period c, if it exists. Pseudo-code for this algorithm is given below.

```
ComputeRetiming(G)
set r(v) = 0 for each node v
for i = 0 to |V|-1
   ComputeDelays(G,r)
   for each node v with D<sub>r</sub>(v) > c
        increment r(v) by 1
ComputeDelays(G,r)
if any node has D<sub>r</sub>(v) > c
      there is no feasible retiming for c.
```

Function ComputeDelays(G,r) computes the delays  $D_r$  in the graph for the given retiming with a similar algorithm. ComputeDelays (G, r)

```
set D<sub>r</sub>(v) = 0 for each node v
for i = 0 to |V|-1
for each edge e=(u,v) with w(e)+r(v)-r(u)=0
D<sub>r</sub>(v) = max(D<sub>r</sub>(v), D<sub>r</sub>(u)+d(v))
Stop if none of the delays have changed
```

If after |V| iterations,  $D_r(v)$  still changes, this means that the graph contains a positive cycle, and there is no solution. In this case, the algorithm takes O(|V||E|) time. In most practical cases this is too slow. Shenoy & Rudell [18] presented a speed-up of this algorithm, which makes the worst-case runtime O(c|E|) time, where c is the length of the smallest cycle with positive delay. They apply a similar speed-up to the retiming algorithm itself.

Given this algorithm, the optimum clock period retiming of a graph can be found by applying binary-search on all the possible clock periods. As an upper bound for the retiming we can take the delay of the graph before retiming.

Leiserson and Saxe also discuss the problem of minimum area retiming under delay constraints. This is the problem where we try to find a retiming with the minimum number of registers for a given target clock period. This problem is also solvable in polynomial time, although the naïve algorithm is very inefficient in practice. Shenoy and Rudell [18] and Maheshwari and Sapatnekar [12] discuss more efficient implementations of this algorithm. However, these algorithms are not as simple and efficient as the Bellman-Ford algorithm for minimum clock period retiming.

Another well-studied problem in retiming is the initial state problem: each register in the circuit has an initial state or power-up condition which may be high, low or don't care. This is the value that the register has at power-up of the circuit. When moving registers around in the circuit, these initial states have to be preserved, that is, the design has to power-up in the same way. When moving registers forward in the circuit (i.e. from inputs to outputs), preserving the power-up state is always possible, because the initial state of the new register can be computed by simulating the gate through which the register has been moved forward. However, when moving registers backwards in the circuit, initial state computation is harder, and sometimes even impossible. The problem of computing initial states for the retimed circuit is NP-hard. Many papers give heuristics to solve this problem, but most are too inefficient and complicated to apply in a practical environment (Touati & Brayton [17], Even, Spillinger & Stok [8], Maheshwari & Sapatnekar [13], and many more).

When mapping for LUT-based FPGAs, applying retiming before mapping to LUTs does not guarantee optimal LUT depth. Retiming on LUTs after mapping however, also does not give optimal LUT depth. Pan and Liu [16] introduce an algorithm that simultaneously maps simple gates to LUTs and applies retiming. Their algorithm guarantees optimal LUT depth, but is very complicated and computationally expensive. Runtime improvements of the algorithm have been presented by Pan and Lin [15], and Cong and Wu [5]. The latter also considered area minimization. Cong and Wu [6] also extended the algorithm to incorporate initial state computation.

In most real-life circuits, there are multiple different clock domains, and registers have secondary signals like clock enables, asynchronous clears, etc. Legl, Vanbekbergen and Wang [9], address the problem of retiming circuits with clock enables, which later is extended to reset signals by Eckl *et al.* [7]. They present an efficient algorithm to compute upper and lower bounds on the retiming of nodes that guarantee that any retiming satisfying these bounds will be legal w.r.t. the clock enables and reset signals. They also

present a heurstic to retime a circuit that consists of multiple related clock domains.

Many papers try to overcome the problem of the lack of estimates of interconnect delay by applying retiming during clustering and partitioning (Cong, Li and Wu [4], Cong, Lim and Wu [2]), during placement (Singh and Brown [19]), and in physical synthesis (Cong and Lim [3]).

There are a number of "pitfalls" in retiming which are rarely discussed in the literature. One of the primary contributions of this paper is in discussing a solution to the problem that does take all of these problems into account. For example, we must deal with multiple clock-domains and secondary signals that introduce the notion of compatible registers. These are a particular issue for FPGAs, which often contain interface logic with different clock domains. Other issues include the interaction with user constraints – a don't touch designation is critical for any serious design in which the designer needs to maintain internal registers for debug purposes - and restrictions on retiming to avoid meta-stability and cross clock domain problems. The latter we will refer to as legality issues in retiming, and we devote Section 4 to the topic. Power-up conditions have been dealt with in the past, but we are not aware of any published papers that cover all the topics necessary for a production guality system simultaneously.

# 3. OUR ALGORITHM

Our retiming algorithm is incorporated in a production tool, which means it has to run fast and be maintainable. These requirements rule out algorithms that combine technology mapping and retiming for two reasons. First these algorithms tend to be time consuming. Second they are so intertwined with the technology mapping algorithm that it will be very hard to get it to work with all the tiny little details that are in a production version of a technology mapping algorithm for multiple different device architectures. Therefore, we have chosen to implement retiming as a separate algorithm that runs on netlists consisting mostly of simple gates and LUTs.

Input to our retiming algorithm is a netlist consisting of simple gates (mostly AND, OR, XOR, and LUTs), primary inputs and outputs, registers, and some gates that are considered hard boundaries for retiming, such as RAM, DSP blocks, and carry chains. Each register has a power-up condition (high, low, or don't care), and zero or more of the following secondary signals: asynchronous reset (clr), asynchronous preset (pre), and asynchronous load with corresponding data (aload/adata),

The ultimate goal of retiming is to move registers around in the netlist such that, after technology mapping, placement and routing, each clock in the design runs at the highest possible speed. Since we have imprecise visibility of placement and routing during synthesis, we can only use a delay model that takes no placement and routing into account. Our delay model is mostly unit-delay, with some special delays for LUT gates and miscellaneous gates like carries for which there is a fast connection on the chip. Even though our experience is that depth does not correlate perfectly to final fmax, improving depth while marginally increasing area in general makes final fmax better. Hence, the goal of our retiming algorithm is to move registers around such that for each clock domain, the clock period after retiming of that clock domain is optimal with respect to our delay model (though evaluation will use final place & route fmax).

The most important constraint for our algorithm is that the netlist after retiming is functionally equivalent to the netlist before retiming. Another important constraint is that the area increases at most 2% on average, and preferably never more than 20%. Other constraints are usability constraints that prevent movement of some registers and that disallow movement of registers over certain gates (e.g. RAM, carry chains). Some more issues are discussed in Section 4

Our algorithm consists of three steps.

- 1. Build a retiming graph from the relevant part of the netlist.
- 2. Find an optimum clock period retiming in the graph, satisfying the given constraints.
- 3. Apply the obtained retiming to the netlist.

Most research papers only address step 2. It is interesting to note that in our implementation, step 2 makes up less than 20% of the code for the complete algorithm. We will discuss each step of the algorithm in more detail in the following sections.

# **3.1 Building the Retiming Graph**

The retiming graph has to model the gates and interconnections that are involved in the retiming, but also all possible restrictions that need to be satisfied. We run retiming per clock domain, and will make separate retiming graphs for each clock domain.

The graph consists of input nodes, output nodes, internal nodes, and a so-called host node. For each combinational gate (simple, LUT, or complete logic cell) that is connected to a register in the current clock domain, we create an internal node in the graph. For each hard boundary gate, we create an input and/or an output node in the graph, depending on whether it feeds or is fed by combinational logic that needs to be included in the graph. For each two nodes u and v in the graph, there is an edge from u to v for every path through only registers from the gate corresponding to u to the gate corresponding to v. The weight of this edge equals the number of registers on the path. Figure 3 shows how a netlist is translated into a retiming graph. Clock and secondary signals of the registers are ignored. The solid bars in the graph denote registers: the number of bars on an edge equals its weight. The existence of node H (the host node) is explained below.

To explain the use of the host node, it is important to know that we use Leiserson & Saxes retiming algorithm to determine a legal retiming for a given clock period. In order to use this algorithm, we need to be able to move registers from output pins to input pins and vice versa in the retiming graph, even though this is not legal in the circuit. We create a host node H that has an outgoing edge to each input node and an incoming edge from each output node. This allows us to move registers from outputs to inputs and vice versa. In the circuit, it is not legal to move registers out of the circuit, or into the circuit, which means that we have to ensure that for our final retiming, r(v) = 0 for all input and output nodes v. This is modeled by giving each incoming and outgoing edge of the node H weight one, and setting the delay of node H to current target clock period. It can be verified that in this way, any feasible retiming can always be changed into a feasible retiming with the same clock period in which the host node and the input and output nodes all have r(v) = 0.

Given a gate and a register on each of its inputs, we can only apply forward retiming if the registers on its inputs are *compatible*. We define two registers to be compatible if



Figure 3: A netlist and its retiming graph.

and only if they have the same clock enable, their asynchronous signals are triggered at the same time, and furthermore, if both registers have an asynchronous load, they load the same data. If the registers on the inputs of a gate are compatible we can move them through the gate and compute the asynchronous signals and power-up state of the new register by simulating the gate. Figure 4 and Table 1 show an example of this for forward retiming through an XOR gate. For each value of an input pin, Table 1 contains the corresponding value of the registers and gates.

To make sure we never try to move incompatible registers forward over a gate, we set lower bounds on the retiming numbers of certain nodes. Legl, Vanbekbergen & Wang [9] and Eckl *et al.* [7] show how this can be done for registers with clock enables, clear and preset. We extend this to handle asynchronous loads as well: Each register is given a class, which basically consists of a clock enable, a list of signals that trigger asynchronous events, and the asynchronous data signal if it exists. Two classes are compatible if they have the same clock enable, the same list of asynchronous trigger signals, and either the same asynchronous data or one has no asynchronous data. Note that compatibility is not an equivalence relation.

Initially when we build the retiming graph, we give each edge (u,v) a list of classes of registers that is formed as



**Figure 4:** Forward retiming with asynchronous secondary signals.

Signal	Α	В	С
Power-up	1	0	1
CLR	0	0	0
PRE	1	1	0
ALD	D	1	D'

**Table 1:** Simulating the asynchronous secondary signals for the netlist in Figure 4.

follows. Suppose when walking on the path from u to v in the netlist, we meet registers  $r_1, r_2, ..., r_n$  in this order. Then the class list on the edge (u,v) will be  $(c(r_1),c(r_2),\ldots,c(r_n))$ , where  $c(r_i)$  is the class of  $r_i$ . With these class lists we compute a so-called maximal forward retiming of the graph, by applying forward retiming as much as possible, but only moving registers forward over a node when all involved registers are compatible. The class lists are adapted during the forward moves. See [9] and [7] for details. When no changes can be made anymore, we have found a maximum forward retiming of the graph. For each node in the graph that has incoming edges with positive weight, but can not be forward retimed anymore because the registers are incompatible, we set the current retiming number as a lower bound on the node. The class lists on the edges will be discarded after this.

In the retiming graph, lower bounds on retiming numbers of nodes are implemented by adding extra edges to the retiming graph. For instance, if we have a lower bound of k on the retiming number r(u) for a given node u, we add an edge from H to u with weight k+1.

For backward retiming, we could have applied the same idea to avoid merging incompatible registers, similar to [7]. However, in this case, it is always possible to insert extra logic to fix the problem of incompatible registers, as is shown in Section 3.3. We found that it would limit the ability for backward moves by too much.

#### **3.2** Finding a good retiming

As said before, a major requirement for the algorithm is that it needs to be fast. Therefore we start with the Bellman-Ford algorithm as presented by Leiserson & Saxe [11], with the additional speed-ups as presented by Shenoy & Rudell [18]. To find the best possible cycle time, we apply binary search on the cycle time.

When the optimal clock period is found we need to find a retiming for this clock period that keeps the area increase under control. We try to minimize the number of register moves, because that has proven to be a good method to control area. Especially backward moves have a potential to increase area, because they may introduce extra logic. Even, Spillinger & Stok [8] apply Bellman-Ford in reverse order, starting from outputs instead of inputs. They claim that it helps keep the total number of backward moves down. We implemented this and found that the algorithm works well for circuits that don't need backward moves, since it will generate none. However, for circuits that require backward moves their algorithm usually needs more backward moves than the original one (this was also noted in [5]). Since it cannot be decided beforehand which algorithm will result in less backward moves, we apply four variations, and pick the best result. The four algorithms are (1) Bellman-Ford, (2) reverse Bellman-Ford, (3) start with forward Bellman-Ford, set retiming to zero for all nodes with r(v) < 0, and apply reverse Bellman-Ford on the partial solution, and (4) start with reverse Bellman-Ford, set r(v) to zero for all nodes with r(v) > 0 and apply forward Bellman-Ford on the partial solution. The noted speed-ups apply to each of these variations, and after Bellman-Ford, we change the retiming of each node such that all input and output nodes have retiming zero. The metric for "best" is min  $sum\{-r(v)*w1 \mid r(v) < 0\} + sum\{r(v)*w2 \mid r(v) > 0\}, where$ w1 and w2 are weights we give to forward and backward moves, respectively.

# 3.3 Applying the Retiming to the Netlist

When the final retiming is found in the retiming graph, it has to be applied to the netlist. Note that at this point we assume the given retiming has r(v)=0 for the host node and all input and output nodes. The retiming is applied to the netlist by moving registers over gates one by one. First the forward retiming is applied to all nodes with the following algorithm. It maintains a stack of all nodes v for which r(v)<0 that have registers on all inputs. Then as long as the stack is non-empty, it pops a node from the stack, calls the function ForwardRetime on this node, and pushes new nodes that require forward retiming on the stack when necessary. The function ForwardRetime(v) forward-retimes the gate in the netlist that corresponds to node v, and increments r(v). It applies simulation to compute the power-up state and asynchronous secondary signals of the new register that is placed on the output of the gate. See also Figure 4 and Table 1.

When the forward retiming algorithm is completed, it is guaranteed that  $r(u) \ge 0$  for all nodes u in the graph. After this, we apply backward retiming in the same way. Like procedure ForwardRetime, we also need a procedure BackwardRetime that applies backward retiming to a gate in the netlist. This function is different from the forward retiming, because it may encounter incompatible registers and furthermore, it is not always possible to compute new power-up states and asynchronous secondary signals after moving registers backwards through a gate. The procedure to solve these problems goes as follows.

Given a gate that needs to be backward retimed, we know that it fans out to only registers. We first merge registers that are equivalent. If the gate still fans out to more than one register after this, then these registers are incompatible, and the gate is duplicated such that each copy has fanout to only one of the registers. All duplicates of the gate will be backward retimed. To backward retime each single output gate, the register is removed from its fanout and registers are inserted at each of the fanins of the gate. If the gate is an AND or OR gate, the new registers get exactly the same power-up setting and asynchronous signals as the original register. It can easily be verified that this gives exactly the same functionality for this gate. If the gate is an XOR or LUT however, setting the same power-up and asynchronous signals would not necessarily give the same functionality. To ensure the same behavior, we do not set different preset and clear signals on the new registers. Instead, we give both registers power-up don't care and no asynchronous secondaries. Then at the fanout of the gate, we insert some extra logic and registers to fix the problem. See Figure 5 for an example.

# 4. LEGALITY AND PRACTICAL ISSUES

There are many issues that make it dangerous or illegal to move certain registers in a design, in a sense that the register moves may change the behavior of the circuit without the designer realizing it, or even being able to detect it. These issues are usually ignored in most research papers, but have to be taken into account in a real-life implementation, and do change the quality of results (by about 50%, as we will see later). We list some of these issues below.

*Registers fed by unrelated clock domains.* Registers in a design may be fed by registers in other clock domains, either directly or through combinational logic. When the



Figure 5: Fixing asynchronous secondary signals in backward retiming.

clock domains are unrelated, it is dangerous to duplicate the register that is fed by the other clock domain: after duplicating it, the two registers may clock in different data at the same clock edge, because they see the data at slightly different times, or because of clock skew. Problems like this are hard to discover for designers, thus we disallow moves of these types of registers.

When a register is fed by a different clock domain, it may go in a meta-stable condition due to setup and hold time violations. To make sure that meta-stable registers don't feed other logic on the chip, the data that comes from the unrelated clock domain is often fed through a sequence of two registers with the same clock, before feeding other logic, to drastically decrease the probability of metastability on the register feeding the rest of the circuit. It is clear that if these synchronization registers are moved away from each other, the absence of meta-stability cannot be guaranteed. Hence we do not allow those moves.

*Input/output registers*. Registers that are directly fed by a pin or directly feeding a pin, are often there on purpose, so that the designer gets deterministic I/O timing. Therefore, we do not move these registers.

*Registers feeding asynchronous signals on other registers.* Duplicating these registers may result in glitches in the asynchronous signal, which may result in unexpected values on the register. Therefore we do not move registers that feed asynchronous signals on other registers.

*Registers feeding registers in another clock domain.* To be safe we do not move these registers either.

The restrictions described above infer that retiming has less flexibility in moving registers around. Since we are potentially over-restrictive on some of the above assumptions and rules, the user has the ability to override our rule by setting a logic option on the register when they know the retiming is safe.

*Non-retimable critical paths.* When a large number of registers cannot be moved due to reasons described above, the delay-critical path in the retiming graph may be formed by a zero-weight path from input nodes to output nodes.

Since these paths cannot be improved by retiming in any way, they are limiting the retiming. We partly solve this problem by removing all edges from the retiming graph that are not on any path with an edge with positive weight (i.e. a movable register). The idea behind this to ignore the critical path that cannot be retimed, and instead concentrate on the next critical path. In the end, this will help fmax, because place & route will deal with less near-critical paths.

*Timing constraints.* Designers can put timing constraints on individual registers. For instance, they can put a multi-cycle constraint from register A to B, meaning that B only clocks in the data from A once every so many cycles. When two registers have different timing constraints, we cannot move them over a gate and merge them, because this would invalidate the timing constraints. We currently disallow movement of registers with individual timing assignments.

*Verification.* Most verification tools that are currently on the market only do static verification of combinational logic. However, this technique is not sufficient to verify that a netlist before retiming is functionally equivalent to the netlist after retiming. This makes it very hard to use retiming when verification is needed.

*Simulation.* The retimed netlist will have the same input/output behavior as the netlist before retiming. This means that simulation on input and output pins will not give any problems. But it will be impossible to simulate registers in the netlist, because registers disappear from the netlist. Similar problems occur when using signal-tap (hardware debug macros supported by the tool) to probe internal nodes.

We support the latter two issues essentially through user logic-options such as don't-touch which can be assigned to named registers in the design, and by outputting a detailed report on registers created and removed by retiming.

#### 5. EMPIRICAL RESULTS

Our results are based on synthesis with Altera's Quartus II V2.2 software, using the Stratix architecture [10].

We run our retiming algorithm at the end of technology independent synthesis, before technology mapping. At this time, the netlist consists mostly of simple gates, which allows for many potential places in the netlist to move registers to. At the end of technology independent synthesis, we have a good visibility of the depth of the final mapping: experiments show that at this point, the depth in two-input gates correlates well to the final LUT depth after mapping. Before running retiming we therefore decompose all simple gates into two-input gates, using the algorithm DMIG [1], which finds a decomposition that guarantees optimal gate depth. The retiming algorithm is run for each clock domain separately. After retiming is finished, we run a clean-up function that makes sure all power-up states and asynchronous secondary signals are legal for the current device, followed by a number of logic minimization steps to remove duplicate registers and redundant logic that may be introduced by retiming.

We compare final fmax results (after placement, routing and timing analysis) when the retiming algorithm is turned on vs. off, along with area and runtime. The data represents average results on a per-design basis with each design run on two initial placement seeds.

The designs used for these experiments are industrial designs that range in size from 12,000 to 56,000 4-LUT+DFF logic elements, and include multiple types of RAM and DSP blocks (dedicated multiply/accumulate hardware). The size of the retiming graph ranges up to 210,000 nodes (gates) in the largest case.

The geometric mean improvement in fmax from our retiming algorithm is 5.0%. However, this is not an accurate metric of the behavior of retiming. Figure 6 contains a gain distribution chart, showing the individual gains over all the designs tested with the algorithm. Though the average improvement is 5.0%, the results are roughly split between winners and losers/ties, with 39 winners gaining 10.2%, 13 ties with no effect, and 15 losers giving up 3.9% fmax. The large number of zeros deserves mention: when the retiming algorithm decides it cannot improve the design we guarantee identical results to the previous case by just not back-annotating the retiming. The negative results represent designs for which the timing model used in retiming predicted a gain, but this turned out to be a loss after place & route.

Designs that have already been well pipelined by the user, and designs which are well-written in general we feel have less to gain from retiming. Designs in the prototype stage, or which have not been analyzed completely tend to get significant gains from retiming. Thus, we believe that this gain distribution is characteristic of retiming in general, and is a crucial part of presenting the results. We have some examples of large designs where over 200% improvements can be achieved with retiming, in one case the critical path goes through a poorly coded multiplier followed by a multiple-bit shift-register. Such a design would significantly improve our reported mean gain. However we have decided that these outliers are not realistic and overstate the gains from retiming, so we do not include them in our benchmarking results here.

For evaluation purposes, we can allow the unsafe retiming discussed in Section 4, and we find that safety constraints block an additional 2.6% fmax gain, which is about 50% over the 5.0% reported for safe retiming. This means that, in addition to generating potentially non-functional circuits,

Retiming fmax gain (Quartus synthesis)



Figure 6: Fmax gain distribution chart for retiming (Quartus II synthesis with and without retiming)

any tools which do not enforce these rules are likely overestimating the gains by a third.

Because of the many other issues with retiming discussed earlier, it is not appropriate to retime a design by default; rather the user should need to turn on the appropriate logic option. Thus, there are some additional benefits to a commercial tool in that a user can try the option, and turn it off in the bad cases. For the stated results, this would mean we could characterize a user's experience as a 50/50 chance of success, with an expected fmax gain of 10.2% on successful designs. This isn't particularly interesting in a research context, but it is worth pointing out as a practical benefit of the gain distribution.

Quartus II software has the ability to un-map an already synthesized and technology-mapped netlist. Thus we can also retime netlists that come from  $3^{rd}$  party synthesis. Benchmarking on such designs shows comparable gains to those shown for Quartus synthesis using this flow.

Some attention was paid to the area problem in our algorithm description. The effect of our retiming algorithm on area is very small. Figure 7 shows a scatter plot of fmax gains vs. area increases. Other than several outliers, we note that the increase in the number of logic elements is within -5% to +5%. The mean change in area is +1%, a very acceptable tradeoff for the performance gains.

The gains from retiming for Altera's Apex architecture are comparable to those seen for Stratix, so our techniques are not architecture-specific.

Runtime is another issue that we brought up early-on. As claimed, the algorithm we have implemented is very fast. The average run-time for retiming is about 2 minutes on an 800 MHz processor, with a worst-case of 11 minutes. This includes the clean-up and minimization steps that are run

Fmax and area tradeoff (Quartus synthesis)



fmax gain

Figure 7: Fmax vs. Area tradeoff for retiming.

after retiming. This translates to about 16% of synthesis runtime and 2% of the overall compile time including place & route. The core algorithm – Bellman-Ford with binary search, averages only about 9 seconds.

All benchmarking here is without any user don't-touch attributes. One would expect the results to degrade somewhat as the problem is constrained.

### 6. CONCLUSIONS

In this paper we have discussed a complete algorithm for retiming, and reported on its positive results. The algorithm draws ideas from a number of different papers, and adds other new ideas, in order to effectively solve fundamental problems such as area degradation and compile time, and combine this with correct handling of power-up conditions, multiple clock domains, secondary signals and legality.

We defined the concept of compatible registers for retiming and gave guidelines on unsafe register moves. To our knowledge nobody has addressed these particular issues in the past, and no paper has addressed all of these issues simultaneously. We also measured the effect on quality of results from properly implementing safe-move constraints in the algorithm, which in our benchmarking represents an additional 2.6% fmax gain on top of the 5.0% results for the safe algorithm, which means that unsafe retiming overstates gains by about one half.

Our retiming algorithm achieves 5.0% mean gains in fmax with a negligible effect on compile time and area. However a further contribution of this paper is to point out that this single statistic is not at all indicative of the behavior of retiming, and we presented a gains distribution chart that gives a better understanding of the overall problem.

#### ACKNOWLEDGEMENTS

Thanks to Vaughn Betz, Guy Dupenloup, and Terry Borer for helpful discussions.

#### REFERENCES

 J. Cong, Y. Ding. "An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs." In *IEEE Transactions on Computer-Aided Design*, vol. 13, pp.1-12, 1994.

- [2] J. Cong, S.K. Lim and C. Wu. "Performance Driven Multilevel and Multiway Partitioning with Retiming", in Proc. *Design Automation Conference (DAC)*, pp. 274-279, 2000.
- [3] J. Cong and S.K. Lim. "Physical Planning with Retiming." In Proc. Int'l Conference on Computer-Aided Design (ICCAD), pp.2-7, 2000.
- [4] J. Cong, H. Li and C. Wu. "Simultaneous Circuit Partitioning/Clustering with Retiming for Performance Optimization. In Proc. *Design Automation Conference* (*DAC*), pp. 460-465, 1999.
- [5] J. Cong and C. Wu. "An Efficient Algorithm for Performance Optimal FPGA Technology Mapping with Retiming". *IEEE Transactions on Computer Aided Design* of Circuits and Systems, vol. 17, no. 9, pp. 738-748, 1998.
- [6] J. Cong and C. Wu. "Optimal FPGA Mapping and Retiming with Efficient Initial State Computation"., *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, no. 11, pp. 1595-1607, 1999.
- [7] K Eckl, J.C. Madre, P. Zepter and C. Legl. "A Practical Approach to Multiple-Class Retiming". In Proc. Design Automation Conference (DAC), 1999.
- [8] G. Even, I.Y. Spillinger and L. Stock. "Retiming Revisited and Reversed." *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 15, no. 3, pp. 348-357, 1996.
- [9] C. Legl, P. Vanbekbergen, A. Wang. "Retiming of Edge-Triggered Circuits with Multiple Clocks and Load Enables". In Proc. Int'l workshop on Logic Synthesis (IWLS), 1997.
- [10] D. Lewis *et al.* "The Stratix Routing and Logic Architecture". Submitted to FPGA 2003.
- [11] C.E. Leiserson and J.B. Saxe. "Retiming Synchronous Circuitry". *Algorithmica*, 1991
- [12] N. Maheshwari and S. Sapatnekar. "Efficient Retiming of Large Circuits." *IEEE Transactions on VLSI Systems*, vol. 6, no. 1, pp. 74-83, 1998.
- [13] N. Maheshwari and S. Sapatnekar. "Minimum Area Retiming with Equivalent Initial States." In Proc. Int'l Conference on Computer-Aided Design (ICCAD). 1997.
- [14] P. Pan. "Performance-Driven Integration of Retiming and Resynthesis". In Proc. Design Automation Conference (DAC), pp. 247-252, 1999.
- [15] P. Pan and C.C. Lin. "A New Retiming-Based Technology Mapping Algorithm for LUT-Based FPGAs. In Proc. ACM/IEEE Int'l Conference on FPGAs (FPGA), 1998.
- [16] P. Pan and C.L. Liu. "Optimal Clock Period FPGA Technology Mapping for Sequential Circuits:" ACM Transactions on Design Automation of Electronic Systems", vol. 3, no. 3, 1998.
- [17] H.J. Touati and R.K. Brayton. "Computing the Initial States of Retimed Circuits", *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 1, pp. 157-162, 1993.
- [18] N. Shenoy and R. Rudell. "Efficient Implementation of Retiming". In Proc. Int'l Conference on Computer-Aided Design (ICCAD). 1994.
- [19] D.P. Singh, S.D. Brown. "Integrated Retiming and Placement for Field Programmable Gate Arrays." In Proc. ACM/IEEE Int'l Conference on FPGAs (FPGA), 2001.
- [20] H Zhou, V. Singhal and A. Aziz. "How powerful is retiming?" In Proc. *IEEE/ACM Int'l Workshop on Logic Synthesis (IWLS)*, 1998.