Incremental Retiming for FPGA Physical Synthesis

Deshanand P. Singh
dsingh@altera.com

Valavan Manohararajah
vmanohar@altera.com

Stephen D. Brown
sbrown@altera.com

Altera Corporation, Toronto Technology Center
151 Bloor St. West, Suite 200
Toronto, CANADA, M5S1S4

ABSTRACT

In this paper, we present a new linear-time retiming algorithm that produces near-optimal results. Our implementation is specifically targeted at Altera's Stratix [1] FPGA-based designs, although the techniques described are general enough for any implementation medium. The algorithm is able to handle the architectural constraints of the target device, multiple timing constraints assigned by the user and implicit legality constraints. It ensures that register moves do not create asynchonous problems such as creating a glitch on a clock/reset signal.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids

General Terms: Algorithms, Measurements, Experimentation, Theory

Keywords: Retiming, Physical Synthesis, FPGA

1. INTRODUCTION

Designs implemented in FPGAs [1] are often dominated by the delay associated with its configurable interconnect. While this phenomenon is also true for ASICs, it is more pronounced for FPGAs because the interconnect contains programmable switches such as pass transistors, tri-state buffers and multiplexers in addition to the metal lines themselves.

One of the most powerful delay optimization techniques is *Sequential Retiming* [3] [4]. This technique moves registers across combinational circuit elements to reduce the length of timing-critical paths. The technique of retiming is ideally suited for FPGA-based implementations because FPGAs are typically fairly register rich architectures that contain a single register for every logic cell in the device.

In this paper, we present a practical retiming algorithm that can be applied to very large circuits with arbitrary timing constraints. The only other published work that addresses these issues is presented in [10]. However, a full description of the algorithm is not provided and a comparison to traditional Leiserson-Saxe retiming is absent. The algorithm presented here also provides a very general framework to handle arbitrary constraints on the retiming moves performed. The constraints include:

- Delay optimization in the presence of user-assigned timing constraints.
- Implicit legality contraints to ensure correct circuit functionality [11].
- Ensuring that newly created registers are distributed evenly across the circuit. This constraint is especially useful for Physical Synthesis flows where newly created logic elements are incrementally placed into the design [8]. The creation of too many logic elements in a localized region may cause significant difficulty in finding an incremental placement.
- FPGA architectural constraints such as carry chains. These chains provide high speed implementations of arithmetic logic; however, signals propagating along the chain must be strictly combinational since most architectures do not allow for registers between chain elements.

The rest of this paper is organized as follows: First, we describe background information about the orginal formulation of speed-optimization by retiming so that we may contrast these techniques with this work. Next we provide a detailed description of the new retiming algorithm. We then present results comparing our algorithm to the original retiming algorithm and results of our algorithm applied to a set of industrial circuits in a Physical Synthesis flow implemented using Altera's *QuartusII* [1] software.

2. BACKGROUND

2.1 Retiming

Sequential retiming is a powerful logic optimization technique for synchronous circuits which uses the property that flip flops can be taken from the outputs of gates and moved to their inputs, or vice versa. Using these moves in combination, one can attempt to maximize circuit speed and minimize area. This technique was first introduced in the early 1980's by Leiserson and Saxe [3] [4].

2.2 Notation and Definitions

Retiming algorithms usually require a unique representation of synchronous circuits. These circuits can be represented using a directed graph of the form $G(V, E)$. V is the

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

set of all combinational cells within the circuit. E is a set of directed edges e_{uv} which denote the connection of cell u to cell v via zero or more registers. Each of the directed edges is associated with a corresponding weight w_{uv}. This weight indicates the number of registers on the connection from u to v.

A retiming of a circuit can be expressed as an integer labeling on each combinational cell. A label $r(v)$ is associated with each cell v. This label indicates the number of registers that are moved from the inputs of the cell v to its outputs. Thus for a given retiming, the number of registers on each wire is given by:

$$w_{r,uv} = w_{uv} + r(u) - r(v) \quad (1)$$

This equation simply expresses that in addition to the original registers on e_{uv}, which is denoted by w_{uv}, $r(u)$ registers are moved onto the wire and $r(v)$ registers are removed.

Given these definitions, the problem of retiming synchronous circuits can then be expressed as finding a label for each combinational cell such that the delay of the longest combinational path is less than a *target* clock period ϕ. This problem can be formally expressed as:

- All retiming labels $r(v)$ must be integers. It is impossible to move fractional numbers of flip-flops from inputs to outputs.

- After retiming, all weights must be non-negative. That is $w_{r,uv} \geq 0$ or:

  $$r(u) \geq r(v) - w_{uv} \quad (2)$$

  This equation exists to ensure that retiming is physically possible or negative numbers of registers may be produced by the retiming algorithm.

- Let P represent a path from $u \rightarrow v$ in a directed graph representation of the synchronous circuit. Every path in the circuit with delay $D(P)$ greater than ϕ must have at least one register along that path.

  $$\begin{aligned} D(P) > \phi \quad &\longrightarrow \quad W_{r,P} \geq 1 \\ D(P) > \phi \quad &\longrightarrow \quad r(u) \geq r(v) - W_P + 1 \end{aligned} \quad (3)$$

  The quantity W_P represents the sum of the weights of the edges along the path P.

This formulation can be solved by a solution to a set of constraint equations. They can be efficiently solved by single source shortest path algorithms such as Bellman-Ford or the algorithm *FEAS* described in [3]. Many techniques have recently been developed for making retiming practical for large circuits [7]. Note that solving these constraint equations does not optimize the clock period, but rather it gives us the answer to a decision problem that asks if the target clock period ϕ can be achieved by retiming. In order to optimize the clock period, a binary search is performed to check individual values of ϕ.

For an FPGA circuit netlist containing n elements, with at most k inputs, the worst-case complexity of this optimal retiming algorithm is $O(n^2 log(n))$. The approaches in [6] [7] dramatically reduce the runtime penalty for retiming algorithms, but the worst-case bound is unchanged.

3. INCREMENTAL MULTI-CONSTRAINT RETIMING

The basic idea behind our retiming scheme is to perform a timing analysis of the circuit and dertermine which registers are on critical or near-critical paths. For each of these registers, r_i, we perform the following actions:

- If the register r_i's input is connected to a critical or near critical path, then attempt to push r_i backward.

- If the output of r_i is connected to a crtical or near critical path, then attempt to push r_i forward.

Before describing our algorithm in more detail, we'll more formally describe timing analysis and notions of criticality.

3.1 Timing Analysis

The most simple timing analysis technique is referred to as *static timing analysis*. This operation is applied to a DAG representation of the combinational portion of the circuit netlist. Each vertex and edge in the graph is annotated with delay information corresponding to the propagation times through combinational elements and the connections between these elements. The *arrival time* at any node, j, is defined as the maximum signal propagation time from any input of the combinational network to the node j. A recursive relation for the arrival times, A_j, can be defined as follows:

$$A_j = max_{i \in FI(j)}\{A_i + delay(i,j)\} + delay(j) \quad (4)$$

The *critical path delay* (T_{crit}) determines the maximum operating frequency of the circuit and is defined as the maximum arrival time over all nodes in the circuit.

The critical path delay is a useful metric for evaluating the effectiveness of the various timing driven techniques described in this paper; however, these techniques usually require knowledge of the parts of the circuit that directly influence the critical path delay so that these areas can be optimized. To accomplish this objective, the concept of the *required time* for a node, i, is defined as the latest time that an input to i can transition without causing the propagation time to any node reachable from i to become critical. In a similar manner to the arrival time, a recursive relation can be defined to find the required times:

$$R_i = min_{j \in FO(i)}\{R_j - delay(i,j)\} - delay(i) \quad (5)$$

The arrival times and required times can be used to calculate the *slack* of a given connection. The slack is defined as the amount of delay that can be added to a given connection without affecting the critical path delay:

$$slack(i,j) = R_j - A_i - delay(i,j) \quad (6)$$

A related measure, termed the *criticality* is defined as:

$$criticality(i,j) = 1 - \frac{slack(i,j) - MinimumSlack()}{max\{T_{crit}, T_{req}\}} \quad (7)$$

Connections on near-critical paths have criticality values close to 1, while non-critical connections have values close to 0.

The procedure described above generates criticality information in the presence of a single user constraint T_{req}. The timing analyzer in Altera's QuartusII's CAD system generalizes this technique to handle multiple user constraints. These constraints include:

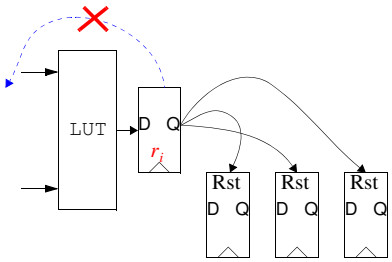


Figure 1: Asynchronous Constraint.

- Required times on each clock domain present in the design.
- Required times from IO pins to registers and from registers to IO pins.
- Implicitly computed required times between related clock domains (ie. clocks and gated versions of the clock).
- Point-to-point timing assignments and multi-cycle constraints. In this way users can inform the timing analyzer that even though a path is fairly long, it may not be critical since it may take multiple clock cycles before its output is needed.

3.2 New Algorithm

The incremental multi-constraint retiming algorithm uses sets of constraints to optimize a circuit in a legal and efficent way. The sets used are:

- TC is a set of all user-defined timing constraints.
- $AREA$ is a set of area constraints for the circuit. This includes a global constraint on the maximum area increase allowed. As described previously, it may also include constraints to ensure that registers are created evenly across the design.
- $ARCH$ is a set of architectural constraints that defines rules for handling carry chains and various restrictions of secondary signals in the target FPGA.
- $IMPLICIT$ is a set of constraints automatically generated to ensure that the circuit functions correctly after the application of retiming. An example of this constraint is shown in Figure 1. In this example a register feeds the asynchronous reset signal of several other registers in the design. Retiming theory allows us to move the source register backwards. However, doing so may introduce a glitch on the signal that feeds the asynchronous lines. This situation could potentially cause disasterous malfunctions. These types of implict legality contraints are described in detail in [11].
- $USER$ is a set of user constraints where they inform our algorithm that portions of logic are not to be touched regardless of the potential benefit.

We also define a set:

$$C_{hard} = AREA \cup ARCH \cup IMPLICIT \cup USER$$

Figure 2 shows the pseudocode for the incremental multi-constraint retiming algorithm. It contains two main loops of K iterations each. Experiments have shown that a value of $K = 32$ works well for most circuits so it is used for all experiments described in this paper. In the first loop, only backward retiming moves are accepted while the second loop attemps only foward retiming moves. Notice that retiming moves are only attempted for registers that have *critical* connections. Experimentation has also shown that we can label a connection, c, as being critical if it satisfies the requirement that $criticality(c) > 0.825$.

In the first loop, we attempt a backward move on all registers that have *critical* inputs. The moves are only accepted if they do not violate any constraint in the set C_{hard}. After all moves are completed, we execute a call to the timing analysis engine in QuartusII. If the worst-case slack has been improved in comparison to the best netlist seen so far, then the current netlist is assigned as the new best netlist. After finishing K iterations of making backwards moves on input-critical registers, we set the current netlist to be equal to the best netlist seen while performing the backwards iterations. Again, the best netlist is the one with the highest worst-case slack. Thus this algorithm is designed to modify a circuit so that it is as close as possible to meeting user timing constraints. Other cost functions such as minimizing the number of paths that cannot meet their timing constraint are also useful to a designer. In fact, our commercial code uses a blend of many different functions to decide on the "best" netlist.

The second loop is identical to the first except that it moves registers that are output-critical forward in the netlist. At the end of this loop, the current netlist is set to be the best netlist found in either the forward or backward pass. It is this netlist that is returned to the QuartusII netlist optimization flow, described later.

The incremental multi-constraint retimer heavily uses the concept of forward and backward register moves. Although these moves are theoretically simple, there are several interesting practical issues that we had to address.

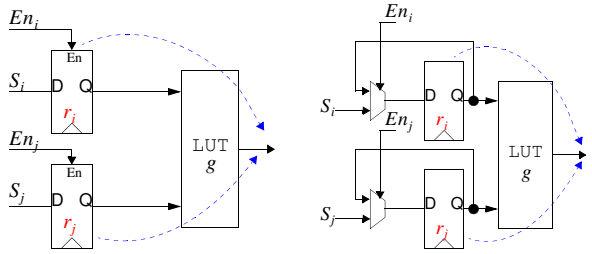
Notice that the worst case complexity of the retiming algorithm is $O(Kn)$, where n is the number of nodes in the circuit. Given that K is a constant, the algorithm has linear time complexity $O(n)$.

Figure 3(a) shows the typical situation present in a forward push of a register r_i. For the forward push to occur, a register has to be present at each input of the combinational element g. If this is not the case, then the push forward simply does nothing and returns. If a register is present at each input of the combinational element, then the push is only legal if the registers all have compatible control signals [2]. In the case shown in the figure, the enable signals to r_i and r_j must satisfy the constraint that $En_i = En_j$.

In some cases, it is extremely beneficial to perform a forward push, but the secondardy signals of r_i and r_j are incompatible. Suppose that we were presented a case such as that shown in Figure 3(a) where $En_i \neq En_j$. In this case we could transform the circuit into an equivalent form shown in Figure 3(b). This transformed circuit uses dedicated multiplexer logic to implement the the clock enable functionality of r_i and r_j. Thus the registers r_i and r_j no longer need enable signals of any sort so the forward push can proceed with no compatibility issue at all. This type of register decompostion is done only when it is needed since

```
proc IncMCRetiming
    call TimingAnalysis(TC);
    BestSlack = MinimumSlack();
    BestNetlist = CurrentNetlist();
    for K iterations do
        foreach register r_i in the circuit do
            if r_i has a critical input then
                push r_i backward;
                if any constraint in C_hard is violated then
                    undo push of r_i;
                end if
            end if
        end for
        call TimingAnalysis(TC);
        if MinimumSlack() > BestSlack then
            BestSlack = MinimumSlack();
            BestNetlist = CurrentNetlist();
        end if
    end for
    SetCurrentNetlist(BestNetlist);
    for K iterations do
        foreach register r_i in the circuit do
            if r_i has a critical output to g then
                push r_i forward across g;
                if any constraint in C_hard is violated then
                    undo push of r_i;
                end if
            end if
        end for
        call TimingAnalysis(TC);
        if MinimumSlack() > BestSlack then
            BestSlack = MinimumSlack();
            BestNetlist = CurrentNetlist();
    end for
    SetCurrentNetlist(BestNetlist);
    return CurrentNetlist();
```

Figure 2: Incremental Multi-Constraint Retiming.



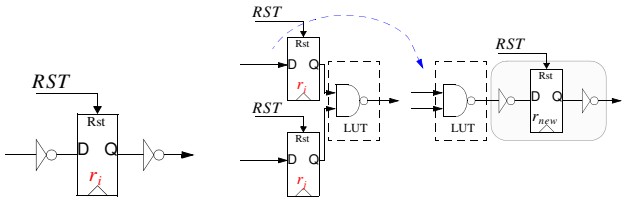
(a) Forward Push. (b) Reg Decomposition.

Figure 3: Forward Push Techniques.

the additional multiplexer logic may produce a significant area penalty if done too often.

An important aspect of retiming that is often overlooked is the computation of reset/power-up states for newly created registers. In Altera's Stratix FPGA, every register in the device[1] is set to logic **0** at power-up and when the register's asynchronous reset signal is asserted. We can simulate a

[1]Register in IO-cells can reset/power-up to either **1** or **0**

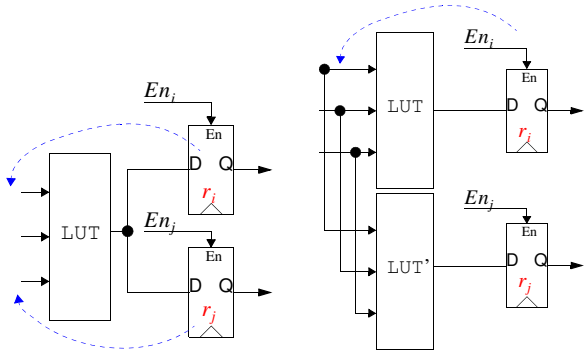


(a) Emulation of Reset-HIGH Regs. (b) Correct Initial State after Forward Push.

Figure 4: Initial State Handling.

register that will reset/power-up to a value of **1** by using the circuit shown in Figure 4(a).

Figure 4(b) shows an example of computing the reset/power-up state of a register pushed across a LUT. In this example, we assume the LUT implements the functionality of a NAND gate and the input registers have reset/power-up states equal to logic **0**. Thus the output of the LUT is set to value of logic **1** when the input registers are reset or at FPGA power-up. We must ensure that this reset/power-up condition is maintained after the forward push is executed. This goal can be accomplished by inserting reset-high circuitry, as shown in Figure 4(a), at the output of the LUT.



(a) Backward Push. (b) Using Duplication.

Figure 5: Backward Push Techniques.

Figure 5(a) depicts the situation in which a backward push is typically performed. A register needs to be present on each fanout of the combinational element to sucessfully perform the backward push. In addition these registers need to have compatible secondary signals and reset/power-up conditions. Again, there are situations where a backward push can be beneficial, but the compatibility issues may prevent the push. In this situation, we simply duplicate the logic element so that the fanouts all have compatible registers. This type of duplication is shown in Figure 5(b). The push backward of register r_i can now proceed with no regard to compatibility issues with r_j or other registers.

The push routines also have special case logic when dealing with logic elements in carry chains. Consider the carry chains shown in Figure 6. If we were to attempt a forward or backward push of a register across a single element in the chain, then the chain would become illegal if the push was successful because registers would be inserted on the special interconnect between cells in the chain. As discussed previously, this dedicated connection can only transmit combina-

Table 1: Incremental vs. Optimal Retiming.

cct	LUTs	IncRt	OptRt	Orig
alu2	197	24ns	23ns	43ns
mult32a	116	12ns	12ns	96ns
s9234.1	461	28ns	28ns	32ns
mm9a	142	68ns	68ns	71ns
s838	167	32ns	32ns	35ns
oc–cordic	1513	4ns	4ns	7ns
elliptic	3602	32ns	32ns	72ns
s38584.1	6281	36ns	36ns	39ns
frisc	3539	33ns	32ns	92ns
hc11	3860	80ns	80ns	120ns
fip–des	15388	32ns	31ns	36ns

tional signals. Since an architectural constraint is violated, the push would eventually be aborted. Thus pushes for elements involving chains would never be accepted. However, many designs are datapath-heavy and the critical path does go though arithmetic elements like adders and multipliers.

We address this situation by moving groups of registers across carry chains as shown in Figure 6. For example when we attempt to move register r_i forward across a carry chain, we ensure that every input to the chain is fed by a register. If this condition can be satisfied and all registers have compatible secondary signals, then the enire group of registers is pushed across the chain in a single atomic operation. Thus the netlist is never illegal at any time. Similarly, backward pushes across carry-chain elements are handled using these group moves.

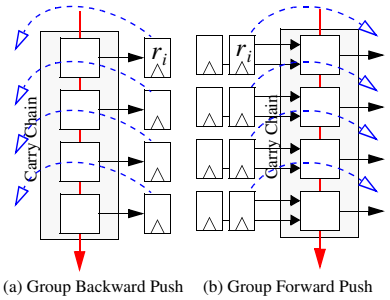


Figure 6: Group Pushes.

4. RESULTS

The first question that needs to be answered is: How does the incremental algorithm compare to the optimal retiming algorithm? Table 1 presents the relevant results. We applied optimal retiming and incremental retiming to a number of MCNC circuits and various free-IP cores that have been synthesized into netlists of 4-LUTs. In this experiment a constant delay is assumed for every logic element and a separate constant is used for each wire in the circuit. The 3rd, 4th and 5th columns of Table 1 show the critical path delays for the incrementally retimined circuit, the optimally retimed circuit and the unretimed circuit, respectively. In each case, the incremental retiming algorihtm matches the performance of the optimal retiming algorithm or is very close. This is an excellent result because the incremental approach can be applied in many more situations than the conventional optimal algorithm.

In addition to experiments using the MCNC circuits with constant delay models, we also measured the performance of our incremental multi-constraint retiming algorithm with industrial circuits targeted to Altera's Stratix [1] FPGAs. Before presenting these results, a brief description of the CAD flow is needed. Our strategy involves a three-step Physical Synthesis approach, as shown in Figure 7, that tightly couples timing-driven circuit optimizations, such as retiming, with the placement step of the FPGA CAD flow. Within the placement phase there is still the freedom to add new elements to the netlist of logic elements, and the routing delays can be accurately approximated for many architectures. In this manner, critical portions of the circuit can be restructured to account for the routing delays.

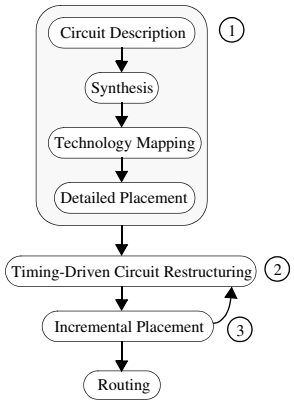


Figure 7: Physical Synthesis Flow.

The first step of the three-step approach executes the conventional FPGA CAD flow of $HDL \rightarrow synthesis \rightarrow techmapping \rightarrow placement$. In the second step, routing delays for every connection are estimated by calculating their fastest possible route. Any timing-driven netlist optimization technique can then be applied to perturb the circuit to reduce the critical path(s). Although numerous netlist optimizations are used collectively in our commercial flow, the experiments in this work deal only with the use of incremental retiming. Every additional logic element or register introduced is given a preferred placement location [8] [9]. These are simply suggestions for "good" physical locations that will optimize timing. Preferred locations for new registers are set to be the same as the logic element that they were pushed across. As discussed previously, we will sometimes create multiplexer logic or perform logic duplication to make a retiming push possible. In these cases, the new logic element is given a preferred location that is identical to that of the original. These preferred locations could form an illegal placement, but we rely on a final fixup stage to remove the illegalities.

The final step of our Physical Synthesis flow occurs after the preferred locations have been generated. The job of the Incremental Placement (ICP) engine is to perturb the preferred locations as little as possible to ensure that the final placement respects all FPGA architectural constraints. These constraints include features such as the limited number of logic elements/inputs/secondary signals per clustered logic block. A detailed description of the incremental place-

ment algorithm used in this study is presented in [8].

Figure 8 shows the results of applying incremental multi-constraint retiming in our three-step CAD flow. This flow is implemented in Altera's QuartusII software and the FPGA used in this experiment is Altera's Stratix device [1] [5]. The circuits used are a set of industrial benchmarks. The incremental algorithm provides excellent performance speedups averaging 7% and reaching 50% in some cases. The runtime is very small in comparison to the entire CAD flow.

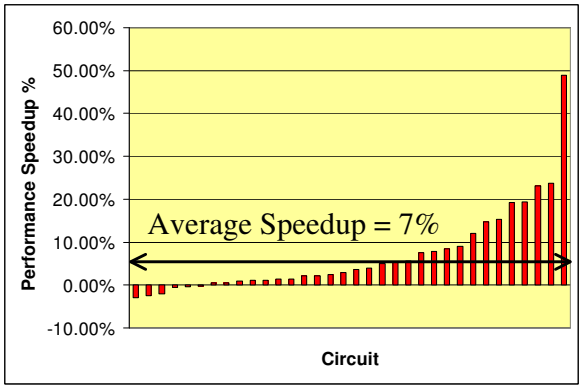


Figure 8: Results on Industrial Circuits.

Finally, we consider the performance of our algorithm in the presence of multiple timing constraints. We use three different types of constraints in this experiment:

- T_{CO} is the maximum propagation time from an active clock edge to a signal change at an output pin.
- T_{SU} is the maximum propagation time from a change on an input pin to reach a register in the circuit.
- T_{MAX} is the maximum propagation time from a register transition to reach another register in the circuit.

Each constraint can have one of two values.

- *EASY* is a value that can be easily achieved.
- *HARD* is a value that is impossible to obtain. An example would be setting T_{MAX} to $1ns$. Currently, our FPGAs do not operate at GHz speeds.

(a) $T_{SU} = HARD$, $T_{CO} = EASY$, $T_{MAX} = EASY$

	%Imp(T_{SU})	**%Imp(T_{CO})**	**%Imp(T_{MAX})**
c1	+58%	+1.3%	+4.1%
c2	+0%	+0%	+0%

(b) $T_{SU} = EASY$, $T_{CO} = EASY$, $T_{MAX} = HARD$

	%Imp(T_{SU})	**%Imp(T_{CO})**	**%Imp(T_{MAX})**
c1	+41%	+1%	+58%
c2	-75%	+7%	+178%

(c) $T_{SU} = EASY$, $T_{CO} = HARD$, $T_{MAX} = EASY$

	%Imp(T_{SU})	**%Imp(T_{CO})**	**%Imp(T_{MAX})**
c1	0%	0%	0%
c2	+4.8%	+5%	+20%

Table 2: Contrained Performance.

Tables 1(a), 1(b) and 1(c) present results for two industrial circuits under a variety of different constraints. Note that we performed experiments with a large number of industrial circuits (≈ 200), but we present only two examples because they clearly illustrate the tradeoffs executed by the algorithm. The entire large set of benchmarks show the same behavior as these two representative examples. Table 1(a) presents the speedups possible when the T_{SU} constraint is set to be the most important and the others are set to values that are easy to meet. The speedups reported compare a run of QuartusII with incremental retiming and the hard T_{SU} constraint to a default run of QuartusII with only the hard T_{SU} constraint. In this way, we report only the speedups due to retiming and not the other parts of the CAD flow that are also trying to optimize for this constraint. Similarly, Tables 1(b) and 1(c) present the results for hard T_{MAX} and T_{CO} constraints respectively. These results show that the behavior of the retiming algorithm changes in the presence of user assigned constraints and that the behavior is to try to improve the part of the circuit with the tightest constraint. In Table 1(b), we show that circuit c2 achieves a 178% T_{MAX} improvement, but with a 75% T_{SU} degradation. This represents the realistic situation in which the required $T_{SU} = EASY$ can still be met, even when increased by a large amount. Notice that in cases of the hard T_{CO} and T_{SU} constraints, one of the circuits could not be improved compared to the default QuartusII run. In these cases, the paths between registers and IOs are already short and it is difficult for retiming to improve them.

5. CONCLUSIONS

We have presented a linear time retiming algorithm that performs almost as well as optimal retiming algorithms. The algorithm directly attacks portions of the design that are deemed critical by Altera's Timing Analysis tool. This concept enables the algorithm to optimize portions of logic that are missing their timing constraints, rather than just optimizing the portions that have the longest delays. The incremental algorithm has benefits in the FPGA environment because it allows for handling device architectural constraints and implicit legality constraints that ensures functionality.

6. REFERENCES

[1] Altera. *Altera Databook*.

[2] K. Eckl, J.C. Madre, P. Zepter and C. Legl. A Practical Approach to Multiple-Class Retiming. *DAC*, 1999.

[3] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry. *Journal of VLSI and Computer Systems*, pages 41–67, 1983.

[4] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[5] D. Lewis, V. Betz et al, The Stratix Routing and Logic Architecture. *FPGA 2003*.

[6] N. Maheshwari and S. S. Sapatnekar. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems*, 6(1):74–83, 1998.

[7] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *ICCAD 1994*, pages 226–233, November 1994.

[8] D. Singh and S. Brown Incremental Placement for Layout-Driven Optimizations on FPGAs. *ICCAD 2002*.

[9] D. Singh and S. Brown. Integrated Retiming and Placement for FPGAs. *FPGA 2002*.

[10] P. Suaris, D. Wang and N. Chou. Smart Move: A placement-aware retiming and replication method for Field Programmable Gate Arrays. *ASIC 2003*.

[11] B. van Antwerpen, M. Hutton, G. Baeckler and R. Yuan. A Safe and Complete Gate-Level Register Retiming Algorithm. In *IWLS 2003*, pages 140–147, 2003.