The Case for Registered Routing Switches in Field Programmable Gate Arrays

Deshanand P. Singh
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
singhd@eecg.toronto.edu

Stephen D. Brown
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
brown@eecg.toronto.edu

ABSTRACT

FPGAs are characterized by a programmable interconnect that contains highly resistive and capacitive elements. While the configurable structure of the interconnect allows for the implementation of arbitrary circuits, it has also become a significant bottleneck for high-speed circuits. Even if there are only a few signal paths that run along long stretches of interconnect, it is these paths that may determine the maximum operating frequency of the circuit.

In this paper we investigate architectural features that could allow us to automatically pipeline the delay associated with long routes without an excessive area penalty. The goal is to reschedule circuit operations in such a way that a signal may use multiple clock cycles to traverse a long route, rather than requiring a single long clock period. This rescheduling would not effect the timing of the visible outputs (no latency is added to the overall system).

Specifically, we analyze the effects of adding a small number of registered routing switches to an FPGA architecture with segmented routing resources. A parameterized FPGA architecture is studied where the percentage of registered routing switches is varied and the speed improvement and area penalty is evaluated. Novel algorithms are presented that allow a circuit to best utilize an architecture with a given percentage of registered switches. We believe that this is the first study that attempts to evauate the tradeoffs associated with switches required in FPGA architectures.

Our experiments indicate that the architectural features introduced can produce significant speedup for high speed circuits without excessive area costs. We believe that these techniques will become increasingly important in the future as deep sub-micron process technologies shrink, and wire delays become even more significant.

FPGA 2001, February 11-13, 2001, Monterey, CA, USA.

1. INTRODUCTION

Designs implemented in FPGAs are often dominated by the delay associated with its configurable interconnect. This phenomenon is also true for ASICs; however, 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.

In a conventional FPGA architecture [1] [13], the length of the longest stretch of interconnect used can be a significant factor in determining the maximum operating frequency. There is no opportunity to traverse the interconnect in multiple cycles. One method of achieving this is to add registered routing switches into the interconnect. While this idea has been proposed before [12] [7], these studies were presented without a detailed exploration on exactly how many of these registered switches are needed to provide good speedup without excessive area penalties. Effective timing driven placement algorithms generate a placement where there are many more short nets in comparison to long ones, because this objective reduces both congestion and critical path delay. Hence we attempt to characterize an architecture such that it has just enough registered switches so that long routes can be pipelined.

The addition of registered switches to the FPGA architecture is only effective if there is a CAD tool to support these features. A novel new algorithm is presented to allow us to map circuits to different architectures with varied registered switch population. The core of our mapping algorithm is the development of a new *Sequential Retiming* algorithm. Retiming is an optimization technique that attempts to pipeline long combinational paths with no effect on the perceived behavior of the inputs and outputs to a given circuit.

The rest of this paper is organized as follows: Section 2 briefly provides background information on the method of Sequential Retiming. Section 3 describes a new FPGA architectural features needed to pipeline long routes. Section 4 provides a description of the parameterizable target architecture used in our experiments. Section 5 describes an algorithm that retimes a circuit such that it can take advantage of the registered switches. Section 6 details our experiments in searching for a new architecture that would allow us to pipeline interconnect delay. Section 7 presents our conclusion and plans for future work.

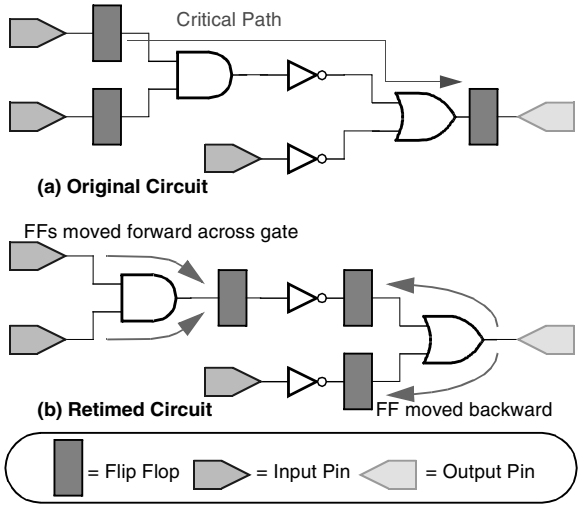

Figure 1: Sequential Retiming

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, without changing the perceived behavior of the circuit. Using these moves in combination, one can attempt to minimize circuit area, speed and/or power. This technique was first introduced in the early 1980's in various works by Leiserson and Saxe [4] [5]. They describe several retiming algorithms to minimize the critical path delay by relocating the registers in synchronous circuits without any change in functionality.

Consider the circuit shown in Figure 1(a). Assuming that the delay of each gate in the circuit is a single time unit, then the critical path delay of this circuit is 3 time units. Retiming theory allows us to reduce the critical path delay by moving flip flops either forward or backward across a gate as shown in Figure 1(b). It is easy to verify that the retimed circuit has the same functionality to the outside world as the original circuit. However we can also see that this circuit contains no path with a delay greater than 1 time unit, and the retiming technique reduced the critical path delay with no changes to the circuit other than redistributing the registers. Thus retiming is suitable to be applied at any step in the CAD flow where a gate-level netlist exists that is annotated with all relevant delay information. The main problem in applying retiming algorithms to FPGA architectures is that once we obtain the final netlist it is not possible to move registers around in an arbitrary manner because conventional FPGAs only contain registers within a logic block. Even if it were advantageous to move a register into the middle of a long route, the architecture could not physically realize such a circuit. We believe this is one of the main reasons that retiming is rarely used in commercial FPGA tools.

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 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. Figure 2 illustrates this concept of the new representation for a cell u connected through a single register to a cell v. The new representation erases the register, and the connection has a directed edge from u to v with weight $w_{uv} = 1$.

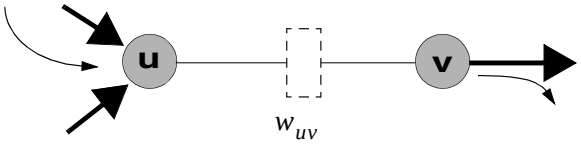

Figure 2: Retiming Circuit Representation.

A retiming of a circuit can be expressed as a labelling 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 integer. 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 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.

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

  The quantity W_P represents the sum of the weights of the edges along the path P. Note that this formulation is slightly different than that described by Leiserson and Saxe. However it is more appropriate for the later discussion on Architecturally Constrained Retiming.

This formulation can be solved by a solution to a set of constraint equations. Since these equations have simple structure, they can be efficiently solved by algorithms such as Bellman-Ford.

3. REGISTERED ROUTING SWITCHES

Previous works [12] on the addition of registered routing switches have based their architectural proposals around the characteristics of the simple retiming algorithm discussed previously. Circuits are retimed, and enough architectural resources are provided to cover the possible outputs of the simple retiming algorithm. For example if the simple retiming algorithm produces an output where six consecutive registers fanin to a LUT, then the architecture is created so that the logic block has 6 extra registers per input.

We feel that this problem should be approached in a significantly different way. We should be able to propose an architecture and create a mapping algorithm that retimes optimally given the constraints of that architecture. In this way we can accurately explore the area-delay tradeoffs associated with various different architectures.

In this section, we will first explore the basic building blocks associated with an architecture that supports registered routing switches. These include registered routing switches themselves as well as extra input registers per LUT. The elements can be added to conventional FPGA architectures in varying quantities and positions. A mapping algorithm is then discussed which evaluates the effectiveness of an architecture which contains these basic elements.

3.1 New Architectural Features

Figure 3 shows an architectural-level description of a registered routing switch. Notice that the switch has both a multiplexer and a register. This allows for routes to be pipelined at arbitrary points or completely unpipelined. The registered switch is shared amongst three tristate driving buffers that connects to other horizontal/vertical segments. The registered switches in our experiments connect only to the ends of the segments. Registered switch connections in the middle provide us with minimal speed gain.

Our circuit-level simulations with SPICE indicate that a registered switch in transparent mode can be made just as fast as a purely buffered switch by sizing the transistors appropriately. This is an important characteristic because this architectural feature should never slow down a circuit in comparison to a conventional architecture.

Figure 4 shows that an extra input register per LUT is added to the architecture. Most modern FPGAs incorporate a hierarchical structure where logic blocks are clustered together in groups and connected by a high-speed local interconnect. The local routing usually takes the form of a fully populated crossbar. This is tremendously useful for retiming purposes. Notice that a single register is placed only the D input of the LUT. However, since the inputs to the LUT are fed from a fully populated crossbar, the inputs to the LUT can be permuted in any way. The truth table in the LUT can also be permuted arbitrarily. These characteristics allows us to register any one of the four LUT inputs. The extra input register is similiar in concept to the variable depth retiming-banks described in [12].

The need for an extra input register per LUT is depicted in Figure 5. If a retiming algorithm needed to move a register from cluster #1 into the long route from #1 to #N, then the algorithm must also move the registers to all of its fanouts to ensure correct operation. Thus a placeholder is needed to enhance the probability that a register can be moved to its fanouts.

An alternative to the LUT-input register is shown in Figure 6. If each logic element is allowed to output both a registered and combinational output, then one could split up the net such that one routing tree distributes a combinational signal and the other would carry the registered version. While this is an attractive solution, it requires us to rip-up existing routes and re-route every time registers are moved into the interconnect. This is not only complex, but the number of routing resources must increase for this technique to be successful since less wires can be shared.

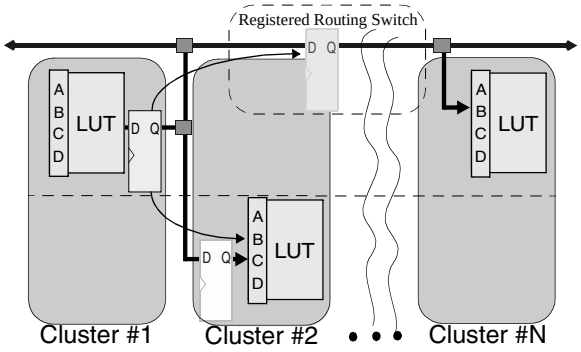

Figure 5: The need for a LUT-fanin register.

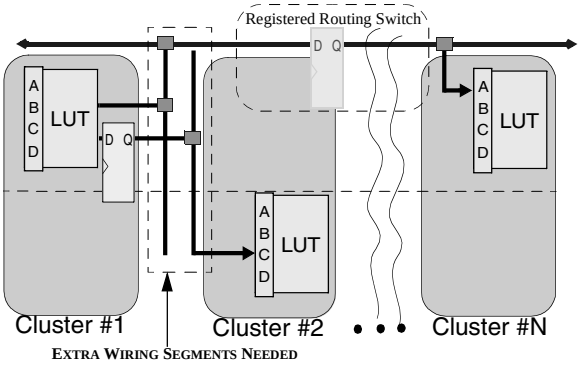

Figure 6: Alternative to LUT-fanin register.

4. TARGET ARCHITECTURE

Figure 7 shows a high-level description the parameterizable target architecture that is used to evaluate the benefits of registered switches. The FPGA contains logic blocks that are clusters of four 4-LUTs. The routing architecture

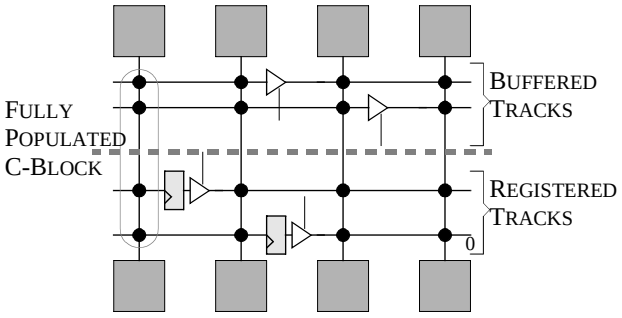

Figure 7: Parameterizable Architecture.

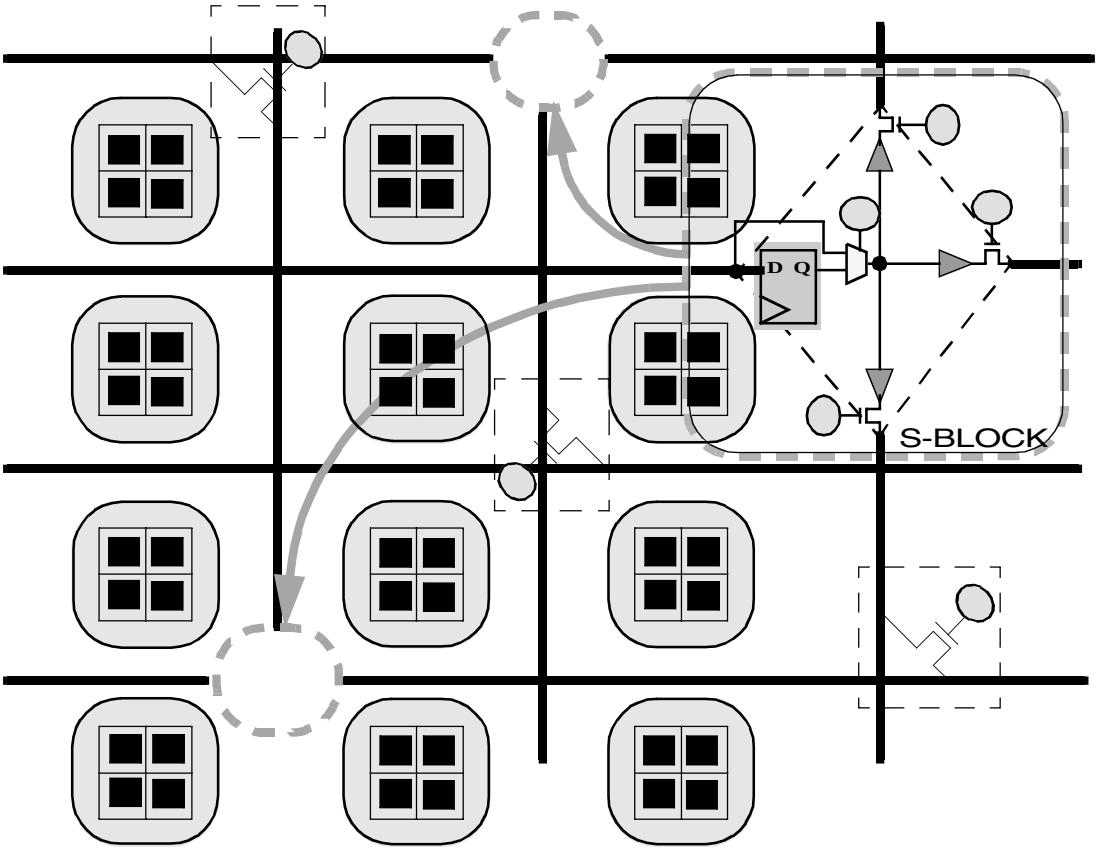

Figure 3: Registered Routing Switches in a Segmented Architecture.

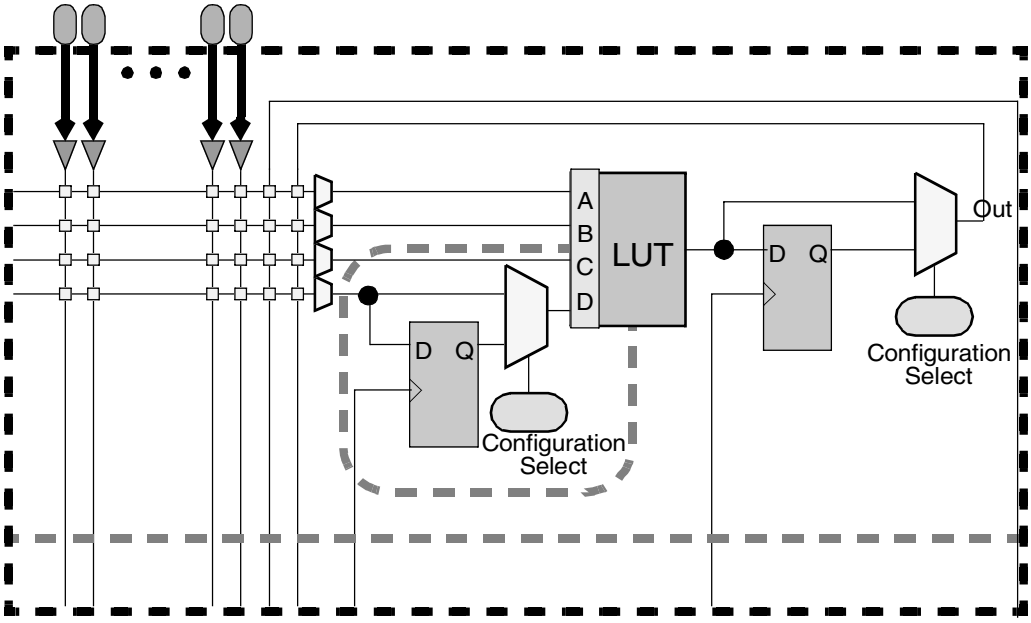

Figure 4: Extra Input Register.

contains only length 4 wiring segments. The topology of the switch blocks is planar: segments are arranged in separate domains called *track planes*. For example, segments in $Track_0$ connect only to other segments in $Track_0$, segments in $Track_1$ connect only to other $Track_1$ segments, etc. This statement holds both for tracks within horizontal or vertical channels and track segments that connect between the horizontal and vertical channels. This style of switch topology was popularized in the Xilinx XC4000 FPGA [13]. The internal points along a segment only connect to other segments via pass-transistor switches. A parameter N indicates the number of track planes in the architecture. The key parameter for the architectural experiments is R, which denotes the number of **Registered Tracks**. All segments on a registered track have registered switches at the ends of the segments. The remaining $N - R$ tracks are referred to as **Buffered Tracks** because these segments have buffered switches at their ends. There is no difference between a registered track and a buffered track except the switches at the ends of the segments. All of the connectivity remains the same. We also assume that registered switches have been sized in such a way that the speed is exactly the same as that of the buffered switches.

5. NEW MAPPING ALGORITHM

The typical CAD flow involved in mapping a circuit to an FPGA involves the steps of *Synthesis*, *Technology Mapping*, *Placement* and *Routing*. We propose a modified CAD flow to map circuits to FPGAs that contain the new architectural features discussed in the previous section. This flow breaks the routing phase into two separate steps:

- **Retiming Aware Routing** - This routing phase attempts to place long connections onto tracks that contain registered routing switches. It is retiming-aware because the router does not actually do the retiming, but rather assigns the nets in such a way so that a retiming algorithm can take advantage of the assignment.
- **Architecturally Constrained Retiming** - This algorithm actually retimes the circuit to achieve a target clock period ϕ. However, the retiming can only be achieved within the constraints of the architecture. For example since we have chosen that there should only be a single LUT-fanin register, then the retiming algorithm must respect this constraint when attempting to solve for the optimal clock period. Registers can only be moved to discrete positions within the interconnect. These constraints must be satisfied to make the retiming valid for an architecture.

5.1 Architecturally Constrained Retiming

The ACR algorithm will be discussed first. Several steps are performed by this algorithm. The input to ACR is a post-route netlist that describes LUTs and the specific routing resources used to connect them. This includes the specific switches and segments used to achieve each connection. It is also annotated with all relevant delay information. *Netlist Conversion* is the first step that is executed to change this input netlist into a form that is usable by the ACR algorithm. After conversion, *New Architectural Constraints* are generated which characterize the limits of register moves in the target architecture.

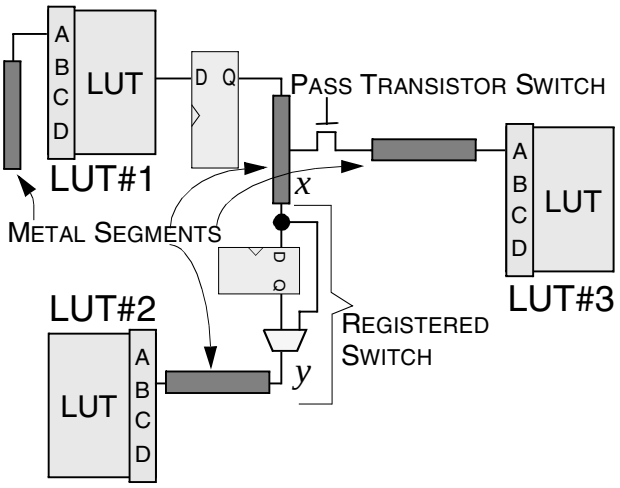

Figure 8: Post Place and Route Netlist.

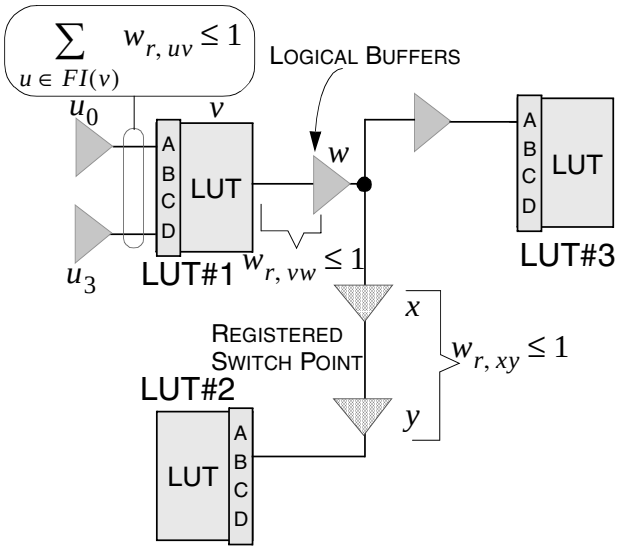

Figure 9: Netlist Representation.

5.1.1 Netlist Conversion

Figure 8 shows a graphical description of the type of netlist that may be passed to the ACR algorithm. Figure 9 shows how it is converted for use by ACR. Notice all delays associated with routing elements are represented by logical buffers. Each LUT contains both fanout and fanin buffers. The weights on the connections between these buffers and the LUT represent the state of the fanout and fanin registers. For example, a weight of 1 on the connection between the LUT and its fanout buffer indicates that the output signal from the LUT propagates through a register before reaching its fanouts. A weight of 0 indicates that the register is bypassed or transparent.

The delay associated with a route through a registered switch is separated by logical buffers at the source and sink of the switch. The weight of the wire in between these two buffers is directly associated with the state of the corresponding registered switch.

5.1.2 New Architectural Constraints

Once the netlist has been converted, ACR must impose additional constraints to ensure that a given retiming does not violate architectural restrictions. The major constraints added by ACR are detailed below:

- **Single Fanin Register Constraint** – The total number of registers on the fanin wires to the LUT must be one or less. This can be expressed in the following manner for every LUT v:

$$\begin{aligned} \sum_{u \in FI(v)} w_{r,uv} &\leq 1 \\ \sum_{u \in FI(v)} (r(u) - r(v)) &\leq 1 \\ \frac{(\sum_{u \in FI(v)} r(u)) - 1}{|FI(v)|} &\leq r(v) \end{aligned} \quad (4)$$

  Note that this derivation assumes that w_{uv} is initially 0 for all fanins, as we assume that the fanin registers are only used by the ACR algorithm.

- **Single Fanout Register Constraint** – The number of registers on the output of a LUT must be one or less. Let v represent the LUT under consideration, and w represent its logical fanout buffer. Then the constraint is expressed as follows:

$$\begin{aligned} w_{r,vw} &\leq 1 \\ r(v) + w_{vw} - 1 &\leq r(w) \end{aligned} \quad (5)$$

- **Registered Switch Constraint** – At every registered switch point along a route, the register may be turned on or transparent. This is equivalent to stating that the number of registers must be one or fewer at the registered switch points. This constraint has exactly the same form as the single fanout register constraint. If x and y represent the source and sink of the registered switch point, then the constraint becomes:

$$\begin{aligned} w_{r,xy} &\leq 1 \\ r(x) - 1 &\leq r(y) \end{aligned} \quad (6)$$

- **Default Constraint** – Every wire ab that is not a registered switch point, LUT-fanin, or LUT-fanout must have no registers. This constraint is expressed as:

$$\begin{aligned} w_{r,ab} &= 0 \\ r(a) &= r(b) \end{aligned} \quad (7)$$

5.1.3 Constraint Satisfaction

After ACR has enumerated all of the architectural constraints, the only task remaining is to determine if these constraints can be satisfied. Figure 10 shows an algorithmic-level representation of the steps utilized by the constraint satisfaction algorithm. The variable C represents a set of constraints that must be satisfied. At the start of the algorithm, C is initialized to contain the basic constraints defined in equation(2), and all of the new architectural constraints discussed previously.

After initialization, C contains all of the architectural constraints to ensure that a retiming respects the architectural constraints of the FPGA. We have not addressed the issue of timing constraints. If we were to enumerate all possible timing constraints, then the memory requirements needed to hold all of these constraints would grow extremely large. This is the case since every path in the directed graph with delay greater than ϕ must be stored. However, there has been extensive work detailing methods to reduce the number of timing constraints [6] [11] that must be stored. While these works show significant reductions, the number of timing constraints can grow extremely large for large circuits. We have found that our largest circuits require hundreds of megabytes of memory to store these constraints, even when using these pruning techniques. Thus we introduce an iterative method of solving the retiming problem that requires much smaller memory requirements.

The basic idea behind our algorithm is to iteratively solve a *partial retiming problem*. On each iteration, the algorithm only constrains the paths that are violated. Consider the simple example shown in Figure 11. On each iteration, the paths with delay greater than 2 time units are constrained

```
proc ConstraintSatisfy( φ )
begin
    C = {Basic Legality Constraints, eq 2}
    C = C ∪ {Single FI Reg Constraints, eq 4}
    C = C ∪ {Single FO Reg Constraints, eq 5}
    C = C ∪ {Registered SW Constraints, eq 6}
    C = C ∪ {Default Constraints, eq 7}
    OldTimingC = {}
    do loop
      1: C = C − OldTimingC
      2: NewTimingC = CombPaths(φ)
      3: C = C ∪ NewTimingC

      4: SOLVE set of constraints.

      5: OldTimingC = NewTimingC
    until Converged or Infeasible
end
```

Figure 10: Constraint Satisfaction Algorithm.

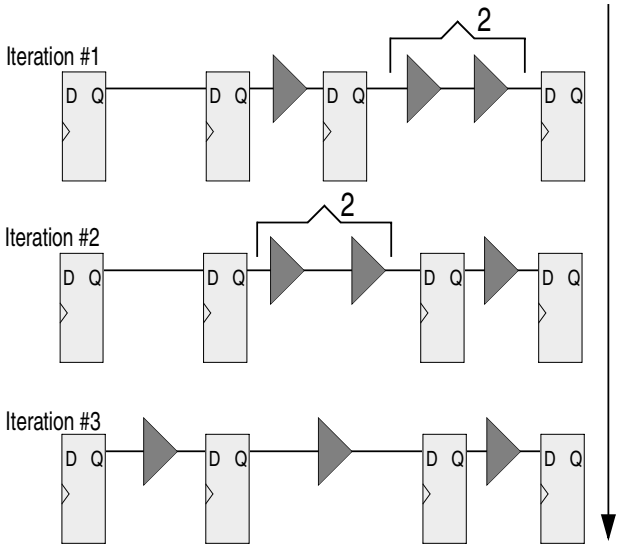

Figure 11: Iterative Timing Constraint Generation.

and the circuit is retimed until convergence. Clearly, the algorithm that solves the partial retiming must exhibit special properties for this scheme to function correctly. Specifically, our partial retiming algorithm makes only forward moves of registers (moves from the outputs back to inputs are allowd if legal), and always makes the minimum possible number of moves.

The ACR algorithm incorporates these ideas of iterative partial retiming. In step 2, the algorithm calls a routine called $CombPaths(\phi)$. This function returns constraints on all of the combinational paths that have a delay that is greater than ϕ in the current netlist. These constraints have the form described in equation(3) and are added to the set C. A constraint satisfaction routine based on the efficient solution of a special case integer programming problem is called, to find if the constraints of the partial retiming problem can be satisfied. If the constraints are satisfied, then the next iteration of the loop removes the the old timing constraints (step 1) and adds constraints for the newly violated paths. It is possible to show that there is a bound on the maximum number of iterations necessary for the algorithm to converge. If more iterations are executed, then the constraints cannot be satisfied. It is also important to note that all of the legality and architectural constraints must be included in every iteration of the algorithm. These ensure that the partially retimed solution is legal.

5.2 Retiming Aware Routing

The objective of a retiming aware router is to make sure that long routes are routed through registered switches. Our current retiming aware router consists of a two-step approach.

5.2.1 Timing Driven Routing

All registered switches in the architecture are treated as if they are simple buffered switches. Registered Switches are used only in transparent mode. Since registered and buffered switches have the same speed characteristics, there is no error in calculating relevant delays. The circuit is then routed using a timing driven router. Our router is based on the Pathfinder [8] algorithm and attempts to minimize the *Penfield-Rubinstein* delay [10]. We also penalize the use of circuitous routes that use pass transistors. This penalty ensures that long routes go through many registered or buffered switch points along its path.

5.2.2 Permuting the Routes

At this point all connections in the circuit have been routed. However there has been no attempt to make sure that long connections are routed through registered switch points. We can now use a special property of planar architectures to modify the routes without changing the current timing, as shown in Figure 12. We can easily permute all of the connections on track 3 with those on track 1 since the configuration of these two planes is exactly the same with the exception of the switch types. Again, it is important to note that registered switches can be made to have the same delay characteristics as buffered switches by sizing appropriately. Note that track planes are not identical because they are usually staggered in some way to enhance routability and to create a tileable architecture; we define two track planes to be *compatible* if they are staggered in the same way.

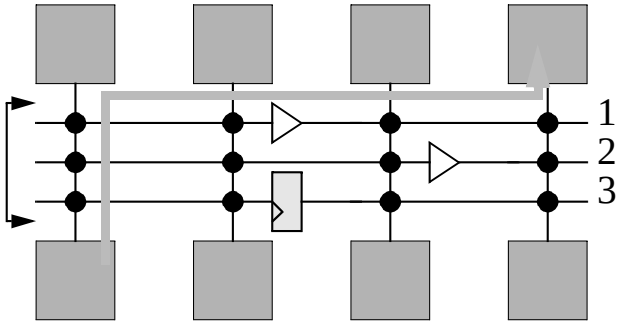

Figure 12: Permuting of Routes.

A track plane permutation can be expressed as:

$$P = \{i, j, \ldots, n\}$$

This notation indicates that after the permutation $Track_0$ will contain all of the routes originally on $Track_i$, $Track_1$ contains the routes originally on $Track_j$, etc. A permutation is legal if and only if $Track_i$ is compatible with $Track_0$, $Track_j$ is compatible with $Track_1$, etc.

Let us assume that a given architecture has R registered tracks $Track_0, Track_1, \ldots, Track_{R-1}$ and N total tracks. After the routes have been completed on the N tracks from the Timing Driven Router; we need an algorithm to compute a permutation that moves long routes onto tracks with registered switches. A simple strategy is shown in Figure 13.

The first stage of the algorithm is called the *Analysis Phase*, which identifies the tracks which have critical routes that could utilize registered switches. This phase operates as follows. We first "pretend" that all buffered switches (BSWs) on buffered tracks are actually registered switches (RSWs). The FPGA now appears to consis entirely of registered tracks. Next we choose one track, $Track_i$, and "pretend" that its RSWs are BSWs. Thus the all of the routes on $Track_i$ have no opportunity to go through registered switch points. The circuit is now retimed while respecting the constraints of this "pretend" architecture. The value of the critical path delay after retiming is assigned to a criticality metric $Crit_i$. Clearly if $Track_i$ contains long routes, then $Crit_i$ will be high since there are no registers available to pipeline the long routes. This procedure is executed for each track with $i = 0 \ldots N-1$.

The second stage is the *Permutation Phase*. This stage attempts to map to a FPGA architecture where the first R tracks are registered tracks. The first step is to sort the tracks by criticality in non-increasing order. Next, the algorithm attempts to permute the most critical track $Track_{c,0}$ with one of the R registered tracks. This can only be done if there is an unassigned registered track that is compatible with $Track_{c,0}$. This process continues in the same way for the remaining tracks.

6. EXPERIMENTAL RESULTS

Figure 14 shows a plot of circuit speedup vs. the fraction of registered tracks $f_r = \frac{R}{N}$ over a suite of various pipelined circuits. These circuits represent a crossection of the benchmarks that were tested. They include pipelined versions of the combinational MCNC benchmark circuits, digital filters, piplined mutlipliers, and microprocessor/alu cores. The size

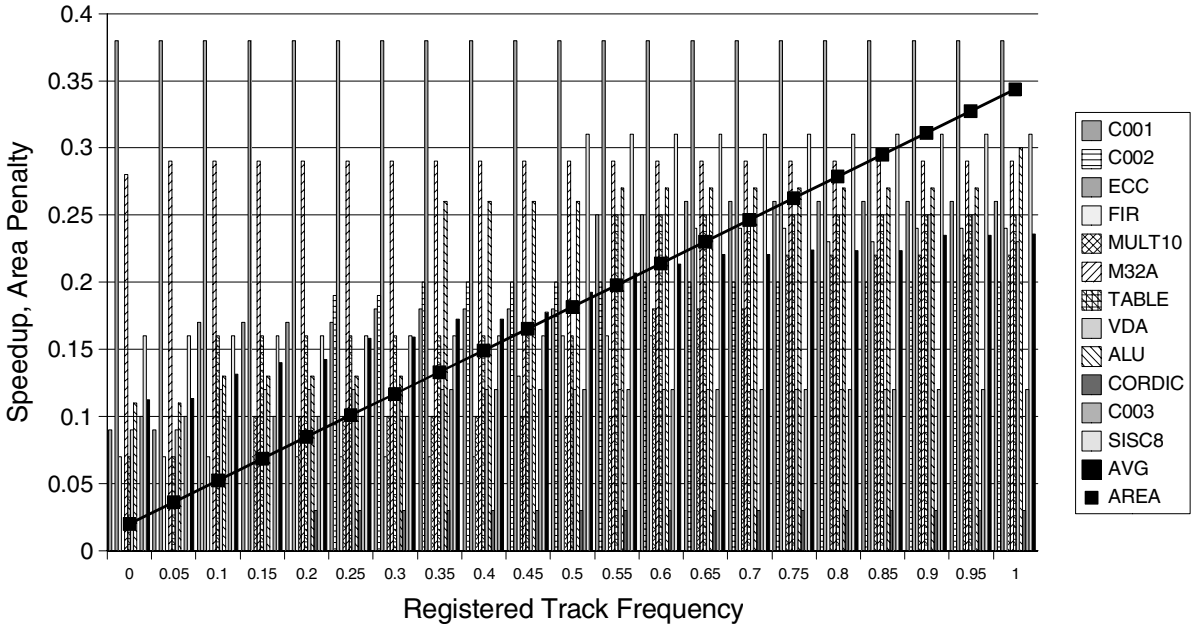

Figure 14: Experimental Results.

```
proc PermuteTracks
begin
    /* Analysis Phase */
    Save type information for all switches
    Replace all BSWs with RSWs
    foreach track t_i do loop
        Replace all RSWs on t_i with BSWs
        Crit_i = SearchForBestClockPeriodWithACR( )
        Replace all BSWs on t_i with RSWs
    end for
    Restore switch type information

    /* Permutation Phase */
    Sort tracks: Crit_{c,0} ≥ Crit_{c,1} ≥ ··· ≥ Crit_{c,n-1}
    R_assigned = 0
    for j = 0 ... n - 1
        if R_assigned < R and
          there exists an unassigned Reg track t_r
          that is Compatible with t_{c,j}
        then
          add t_{c,j} → t_r to the permutation
          R_assigned = R_assigned + 1
        else
          let t_b represent any available, compatible
              buffered track
          add t_{c,j} → t_b to the permutation
        end if
    end for
end
```

Figure 13: Track Permutation Algorithm.

of these circuits ranged from 100 → 7500 LUTs. These circuits were mapped to LUTs using FlowMap [3] and placed using VPR [2]. For each circuit, the smallest FPGA in which the circuit could acheive a fit is computed. The number of tracks used to run the experiment is 20 percent greater than the minimum required. This test exercises our algorithm with low stress routes. When there are relatively few wiring tracks available, there is a greater chance of finding extremely long circuitous routes. The performance of these circuits could be greatly enhanced by moving registers along these routes. However, we feel that analyzing these results would be overly optimistic. This testing methodolog also allows for a size independent comparison of small and large circuits. If one small circuit were mapped into a large FPGA, then virtually all nets could be routed on registered tracks. This would provide misleading results as very small f_r ratios could appear to provide tremendous speedups.

Since we are using a retiming algorithm with the new architecture experiments, one would expect that some portion of the speedup is due just to the retiming algorithm and not because of the new architectural features. To compensate for this phenomenon, we define the speedup in a unique manner. A quantity named T_{base} is defined as:

$$T_{base} = \textbf{BestClockPeriodwACR(Orig Arch)} \quad (8)$$

This quantity refers to the critical path delay after ACR has been run with the original architectural constraints. ACR runs without being able to move registers into the interconnect, or utilizing the extra fanin register. ACR will actually produce modest speed improvements for selected circuits even without architectural improvements. Next T_{retime} is calculated by running ACR constrained to the new architecture.

$$T_{retime} = \textbf{BestClockPeriodwACR(New Arch)} \quad (9)$$

The speedup can then be given by:

$$T_{speedup} = \frac{T_{base}}{T_{retime}} \quad (10)$$

An estimate of the area penalty (the line graph in Figure 14) is included with our results. This area estimate is obtained by summing the silicon areas required by individual switches over the FPGA. The silicon area of the individual switches was obtained from layouts in a 0.35 micron process.

The results in Figure 14 show that for $f_r = 0.25$, the speedups obtained are generally in a bin from $12 \rightarrow 25$ percent. The area penalty slightly exceeds 10 percent.

7. CONCLUSIONS

The use of registered switches within the routing fabric combined with an architecturally constrained retiming algorithm can be extremely beneficial to many circuits. We do not claim that this is a general optimization technique, but rather that it can improve the speed performance of *high-speed* pipelined circuits. Specifically a lower bound [9] on the best obtainable speed of a circuit implemented after retiming is given by:

$$T_{best} = \max_{C} \frac{\sum_{v \in C} delay(v)}{\sum_{uv \in C} regs(u \rightarrow v)} \quad (11)$$

This metric provides us with the largest delay-to-register ratio around any sequential cycle in the circuit. Clearly, retiming cannot do any better than evenly distributing the delays around the cycle. Let T_{route} be the delay associated with the worst route in the circuit. If T_{route} constitutes a significant portion of T_{best}, then it is likely that the registered routing switches can help to distribute the delay associated with the poor routes. We have observed that circuits that exhibit the characteristic $\frac{T_{route}}{T_{best}} \geq 0.5$ usually can benefit from the registered switches.

We speculate that better results can be obtained by considering a more integrated approach to retiming aware routing. The router should be able to automatically assign long/critical routes to the registered tracks. This technique would eliminate the need for a post processing step that considers routing permutations. Better results could be obtained because several long routes could be assigned to one track plane rather than spread out over several planes. Thus greater speedup might be attained at lower values of f_r. Our technique of permuting routes is only valid for a subset of planar architectures. A true retiming aware router would target any architecture by adjusting the cost function so that long routes benefit from going through registered switch points. The great challenge of this approach is to ensure that the worst-case critical path (ACR cannot help) is not increased in comparison to a conventional router.

8. REFERENCES

[1] Altera. *Altera 2000 Databook*. Available from: http://www.altera.com/html/literature/lds.html.

[2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[3] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on CAD*, pages 1–12, Jan 1994.

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

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

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

[7] A. Marshall, J. Vuillemin, T. Stansfield, I. Kostarnov, and B. L. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, Monterey, CA, Feb. 1999.

[8] L. McMurchie and C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs, 1995.

[9] M. C. Papaefthymiou. Understanding retiming through maximum average-weight cycles. In *3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 338–348, July 1991.

[10] J. Rubinstein, P. Penfield, and M. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on CAD*, 2(3):202–211, 1983.

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

[12] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek and André De Hon. HSRA: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 125–134, Monterey, CA, Feb. 1999.

[13] Xilinx. *Xilinx 2000 Databook*. Available from: http://www.xilinx.com/partinfo/databook.htm.