Integrated Retiming and Placement for 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

Retiming is a synchronous circuit transformation that can optimize the delay of a synchronous circuit by moving registers across combinational circuit elements. The combinational structure remains unchanged and the observable behavior of the circuit is identical to the original.

In this paper, we address the problem of applying retiming techniques to circuits implemented in Field Programmable Gate Arrays (FPGAs). FPGAs contain prefabricated and configurable routing elements that allow us to easily implement a variety of circuits. However this interconnect contributes greatly to the overall delay in the implemented circuit. If a circuit is retimed prior to the placement and routing phases of the CAD flow, then it has no information about the delays introduced by the configurable interconnect. Our fundamental experiment is to determine whether there are any gains in tightly coupling retiming and placement so that the retiming algorithm has some estimate of the routing delays.

Specifically, we introduce a post-placement retiming algorithm that understands how to take advantage of FPGA architectural features. This retiming algorithm may introduce extra registers into the circuit. These new registers need to be placed in some location in the FPGA. Retiming register placement is accomplished by a novel incremental clustering and placement algorithm. The incremental algorithm builds upon the placement of the non-retimed circuit to intelligently sift in the newly-introduced registers.

In addition, we explore making the placement algorithms "retiming aware." These placement algorithms try to place logic blocks in such a way that the subsequent retiming produces better speed results. These techniques include the identification of retiming-critical cycles during placement.

Our experiments show that the integration of retiming with placement results in 19% better clock periods in comparison to the application of retiming before the place and route steps.

FPGA'02, February 24-26, 2002, Monterey, California, USA.

1. INTRODUCTION

Designs implemented in FPGAs [1] [15] 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, tristate buffers and multiplexers in addition to the metal lines themselves. This trend is ever increasing as deep submircon technologies shrink, and the wire delays themselves increase.

One of the most powerful delay optimization techniques is *Sequential Retiming* [7] [8]. This technique moves registers across combinational circuit elements to reduce the length of timing-critical paths. Circuit optimization techniques, such as retiming, are typically applied to a gate-level netlist before the placement and routing steps of the CAD flow. We question if this is the correct approach, since the circuit delay is dominated by the flexible FPGA interconnect.

In this paper, we compare the conventional application of retiming at the gate level with a novel retiming process that happens after placement. Application of sequential retiming after the placement step gives us reasonably accurate estimates of the connection routing delays; however, it introduces many new challenges. For example, retiming may introduce several additional registers in the netlist. We need to find an appropriate place for these registers, but the placement phase has already been completed. If the placement phase were to be rerun, then we would pay an extremely high price in terms of compile time. If the compile time is not an issue, this process still does not guarantee convergence since the new placement may be completely different than the original placement. Hence a totally different retiming might be needed.

To overcome these problems, we introduce a new retiming algorithm along with an incremental clustering and placement tool. The new retiming algorithm tries to change the post-placement netlist as little as possible. If this minimally placement-disruptive retiming algorithm does find it necessary to create additional registers in the netlist, the incremental clustering and placement tool is used to find places for these retiming registers. This process may involve moving non-critical sections of logic so that we can place delay critical registers in their preferred locations.

We also identify certain circuit configurations that make post-placement retiming ineffective, and show how FPGA placement algorithms can be modified to be aware of these retiming-limiting cases.

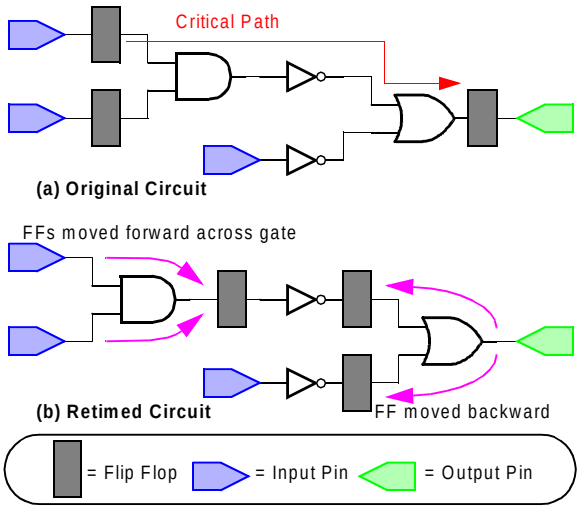

Figure 1: Sequential Retiming

The rest of this paper is organized as follows: Section 2 briefly provides background information on the method of Sequential Retiming. Sections 3,4,5 and 6 describe the algorithms in the order that they appear during a typical CAD flow. Section 3 describes circuit structures that limit the benefit from retiming. Section 4 shows how the placer can be improved to reduce these potential bottlenecks. Section 5 describes the minimally placement disruptive retiming algorithm. Section 6 introduces a high-level description of the incremental clustering and placement algorithm. Section 7 details our experiments and show a cross-section of the results. Section 8 presents our conclusion and plans for future work.

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 maximize circuit speed and minimize area. This technique was first introduced in the early 1980's in various works by Leiserson and Saxe [7] [8]. 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 if we assume that the initial state on each register is 0. 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.

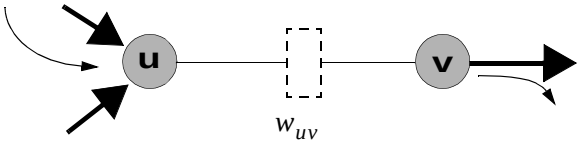

Figure 2: Retiming Circuit Representation.

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 deletes the register, and the connection has a directed edge from u to v with weight $w_{uv} = 1$.

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:

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

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:

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

  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.

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

  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. Since these equations have simple structure, they can be efficiently solved by single source

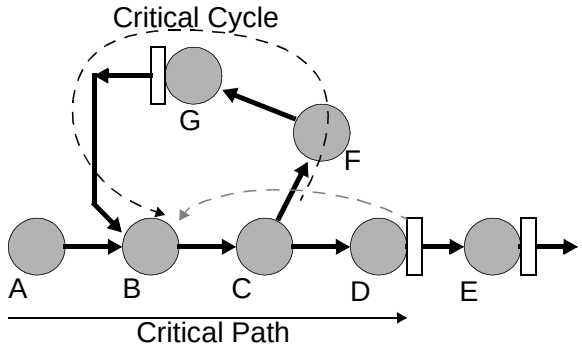

Figure 3: Critical Cycles.

shortest path algorithms such as Bellman-Ford. Many more efficient techniques have recently been developed for making retiming practical for large circuits [14]. Note that solving these constraint equations does not optimize the clock period, but rather it gives us the answer to a binary 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.

A fundamental retiming concept used in this paper is that the number of registers around any cycle in the retiming graph cannot be changed by the application of sequential retiming. The registers may be redistributed around the cycle but the total number cannot be changed. Consider any cycle of the form $v \longrightarrow \ldots \longrightarrow v$. At the beginning of the cycle $r(v)$ registers are added, and at the end $r(v)$ registers are taken away. Hence the number of registers on the cycle must remain unchanged. A more formal proof is given in [7] [8].

3. CRITICAL CYCLES AND CYCLE SLACK

Our experiments with retiming have shown that retiming solutions are often limited by critical cycles in the netlist. This situation is depicted in Figure 3. The critical path *A-B-C-D* can be reduced by moving the registers at D and E backwards. However there is no way of reducing the delay around the cycle *B-C-F-G*. Thus it is this cycle that may limit the performance of retiming. In a conventional placer, there is no reason that the delay around the cycle *B-C-F-G* should be minimized. As long as it is less than the critical path, the placer can organize the nodes on the critical cycles to optimize wirelength or other objectives. However if the placer had knowledge that the critical path *A-B-C-D* could be broken up, then it may be able to reduce the critical cycle. Hence, our objective is to develop a strategy that would give the placer awareness of the retiming possibilities.

Figure 4 shows another factor that limits retiming effectiveness. A retimed circuit may have multiple near-critical paths. In the example, the critical path *A-B-C* could not be retimed because moving the register at the output of C backward would cause the near-critical path to become critical. Again, if the placement engine understood that the circuit would be retimed then it might be able to reduce the delay of the near critical path *D-E*.

To provide the placer with knowledge of the retiming solutions, a special graph is constructed from the netlist being mapped. This graph will be termed the cycle rate graph. To our knowledge, this type of representation was first discussed in [13]. However, it has since been used in many applications [11] [12] which couple retiming with other optimizations.

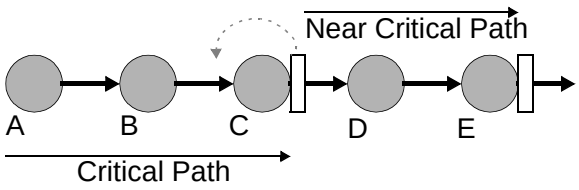

Figure 4: Near Critical Paths.

Let ϕ represent the clock period after the application of retiming. A retiming netlist can be transformed into a cycle rate netlist by creating a new graph with the same vertices as the retiming netlist. For every edge e_{uv} in the retiming graph, we create an equivalent edge in the cycle rate graph. However the weight of the edge in the cycle rate graph is equal to $w_{uv} * \phi - d(v) - d(uv)$, where $d(v)$ is the combinational delay of the node v, and $d(uv)$ is the combinational delay of the edge from u to v. A special vertex called the host is added to the cycle rate graph. The host node is used to model the connections to the external system for the circuit under consideration. Otherwise the retiming algorithm may reduce the critical path while increasing clock-to-output or input-to-clock delays. Zero weight edges are added from the host to each primary input. Edges with the weight of ϕ are added from each primary output to the host vertex. The idea behind the cycle rate graph is that every register allows for ϕ units of combinational delay. Every vertex v uses up $-d(v)$ units of combinational delay. If there is a delay associated with e_{uv}, then it also consumes $-d(uv)$ units of combinational delay. Since the number of registers around any cycle remains unchanged, the sum of the weights around any cycle C gives us $\phi * \Sigma_C w_{uv} - \Sigma_C d(v) - \Sigma_C d(uv)$. This quantity can be thought of as the total amount of combinational delay allowed by the registers, minus the total amount of combinational delay used. Setting this quantity to zero provides us with the balance point where the registers allow as much combinational delay as used by the combinational elements. Equation 4 shows that this equality can be used to solve for a bound on the target clock period. ϕ is limited by the delay around the cycle divided by the number of registers available. This quantity is known as the *delay-to-register ratio* (DRR) for the cycle. The operating speed of a circuit is limited by the largest DRR for any cycle in the circuit.

$$\phi \Sigma_C w_{uv} - \Sigma_C d(v) - \Sigma_C d(uv) = 0$$
$$\phi = \frac{\Sigma_C d(v) + \Sigma_C d(uv)}{\Sigma_C w_{uv}} \quad (4)$$

Notice also that all paths from primary inputs to primary outputs participate on a cycle because of the connections to the *host* vertex. The connections from the primary outputs to *host* have a weight of ϕ because we assume that the circuit will allow us up to ϕ units of combinational delay before it samples its the values at the primary outputs.

These concepts are easiest to understand by looking at an example. Figure 5 shows a retiming graph, and Figure 6 shows the cycle rate graph that corresponds to this netlist. Assume unit delays for each gate in the netlist, and that each edge has a delay of 0. Note that the cycle rate graph has all

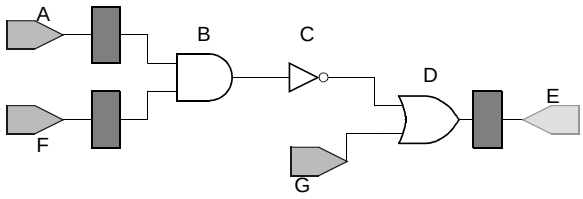

Figure 5: Cycle Rate Netlist.

of the vertices in the retiming netlist. It also contains the host vertex connecting to the inputs and outputs. Examine the edge e_{AB} in the cycle rate graph. In the corresponding netlist, there is a single register connecting from A-to-B. Thus $w_{AB} = 1$, and the weight of the edge in the cycle rate graph is $w_{AB}\phi - d(B) = \phi - 1$. Consider the cycle *A-B-C-D-E-Host*. The sum of the edge weights around this cycle is $3\phi - 3$. Setting this quantity to 0 and solving for ϕ indicates that this cycle may be clocked at a single unit delay if we can successfully retime the circuit. Indeed this is true as we can move the leftmost registers one unit to the right and the rightmost registers one unit to the left.

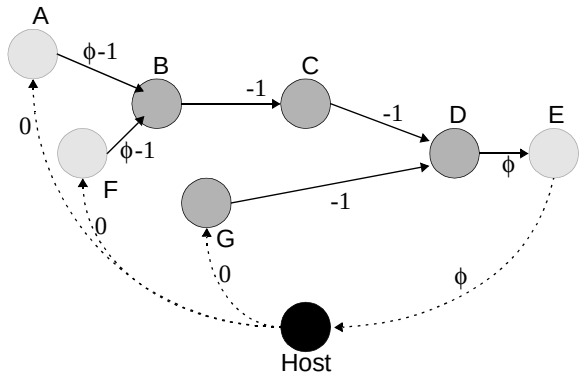

Figure 6: Cycle Rate Graph.

Thus far we've looked at individual cycles and found the DRR that limits the value of ϕ. To find the maximum DRR (MDR) cycle in the cycle rate graph, we re-express the problem as one of finding the minimum value of ϕ that does not cause a negative weight cycle to be created in the graph. For example, in Figure 6 a value of $\phi = 0.5$ would cause the cycle *A-B-C-D-E-Host* to have a negative weight $3 * 0.5 - 3 = -1.5$. A simple technique for finding negative cycles is to run a single source shortest path algorithm, such as Bellman-Ford, on the graph. If the solution does not converge, then a negative weight cycle is present in the graph. However, the techniques described in [4] [14] can be used to more efficiently detect the presence of negative weight cycles.

The techniques described above allow us to find the cycles that limit our target clock periods. To apply it to the placement algorithm, we now define a concept called the *cycle-slack*. We first find the limiting value of ϕ using the techniques described above. For this value of ϕ, the *cycle-slack* of a connection is the maximum amount of delay that can be added to the connection without creating a negative weight cycle. Suppose that we target $\phi = 1$ in Figure 6, then we can tolerate an extra one unit of delay on the connection from G-to-D without creating a negative cycle. Hence the $CycleSlack(GD) = 1$. Notice that the cycle-slack for connections on the critical cycle is 0. If the connection delay from B-to-C were increased at all, then the cycle *A-B-C-D-E-Host* would have a negative weight. It is these cycle-slacks that can be used to inform the placer about which connections are critical, given that we will perform retiming after placement.

We do not know of an efficient (less than $O(n^2)$) algorithm that solves the cycle slack problem exactly. Here we give an overview of an approximate algorithm for solving the cycle slacks. Consider finding the cycle slack for a connection e_{uv}. Choose an arbitrary source vertex s in the cycle rate graph. Find the shortest path from s to u and call this P_{su}, find the shortest path P_{vs} from v to s. Figure 7 depicts this situation. The *cycle-slack* for e_{uv} can be approximated as $P_{us} + w_{uv} * \phi - d(uv) + P_{vs}$. The situation depicted in Figure 7 is typical, but the paths P_{us} and P_{vs} may intersect at a common vertex c as shown in Figure 8. In this case the cycle slack is computed for the cycle *c-u-v*. This scheme only an approximate solution to the cycle slack problem because we only look for cycles that contain s. We can improve our approximations by repeating this procedure for other randomly chosen source vertices s and retaining the smallest slack computed for the different sources. Note that the *host* node is always one of the source vertices considered since many cycles contain this vertex.

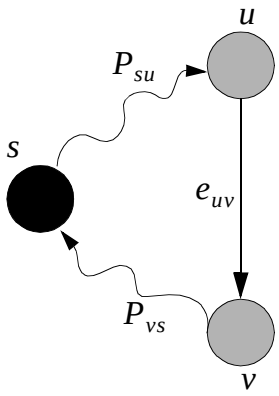

Figure 7: Finding the Cycle Slack.

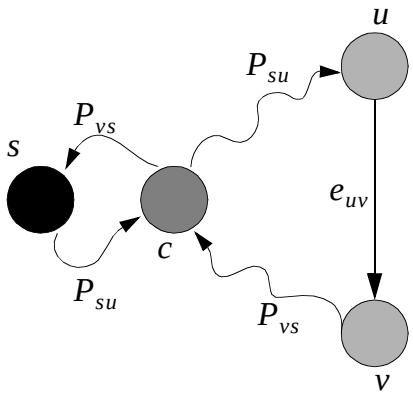

Figure 8: Intersecting Paths.

Cong and Lim [6] use a strategy similar to *cycle-slacks*

to couple partitioning, floorplanning and retiming for ASIC designs. However, their reported procedure behaves differently than our approach for circuits that contain critical sequential cycles.

4. RETIMING AWARE COST FUNCTION

We add retiming awareness to the VPR [2] placement and routing tool for FPGAs. VPR is based on simulated annealing optimization of wirelength and timing cost functions. The VPR timing cost uses connection criticalities to weight the delay of connections. Critical connections are encouraged to be placed closer together. VPR uses the formula in Eq. 5 for converting slacks into criticalities. Note that T_{crit} represents the delay of the critical path in the circuit.

$$Crit_c = 1.0 - \frac{Slack(c)}{T_{crit}} \quad (5)$$

We use an adaptive strategy to convert cycle slacks into criticalities as shown in Eq. 6.

$$Crit_c = 1.0 - \beta CycleSlack(c) \quad (6)$$

The value of β controls the criticality distribution as shown in Figure 9. This figure shows the cycle slack criticalities for two different values of β computed for a single benchmark circuit. Larger values of β flatten the tail section of the criticality distribution. We choose a value of β that ensures that the tail of the criticality profile is relatively flat. This ensures that the placer is not "confused" by too many critical connections. The value of β is initially set to $\frac{1}{AverageSlack}$, and it is iteratively adjusted by examining the criticality profile histogram associated with previous β values. The goal of the adjustment is to constrain the number of edges that are 90%-100% critical to no more than 5% of the total number of edges.

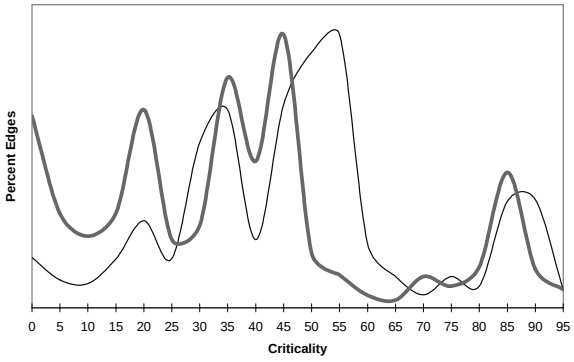

Figure 9: Criticality Profile.

5. MINIMALLY PLACEMENT DISRUPTIVE RETIMING

Once the placement is completed, the next step is to actually do the register moving to retime the circuit. For a given clock period, several different sequences of register moves may achieve the target period. In this section, we develop techniques to find a sequence of register moves that will minimally disrupt the post-placement netlist. This step will make the job of the incremental clustering and placement algorithm much easier.

5.1 Costing Logic Duplication

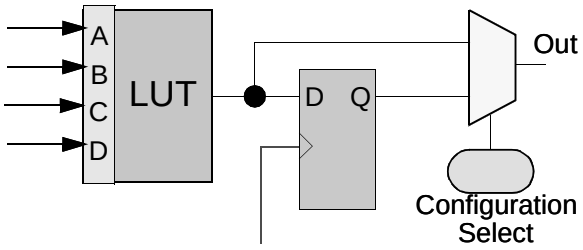

Figure 10: Simplified FPGA Logic Block

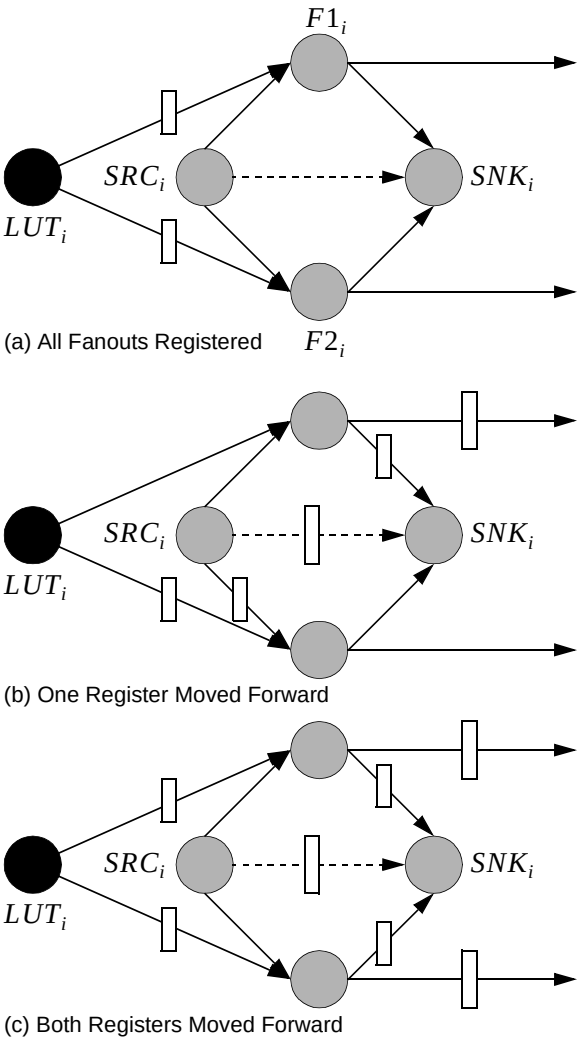

Figure 11: Costing Logic Duplication.

The first challenge is to model the registers at the outputs of FPGA logic blocks correctly. Figure 10 shows a **simplified** version of the logic block used by most commercial FPGA architectures. The block contains a lookup-table with an optional flip-flop to register the output if needed. Our model assumes that only one signal can be output from a logic block. Therefore if both the combinational and registered version of the signal at the lookup table is needed,

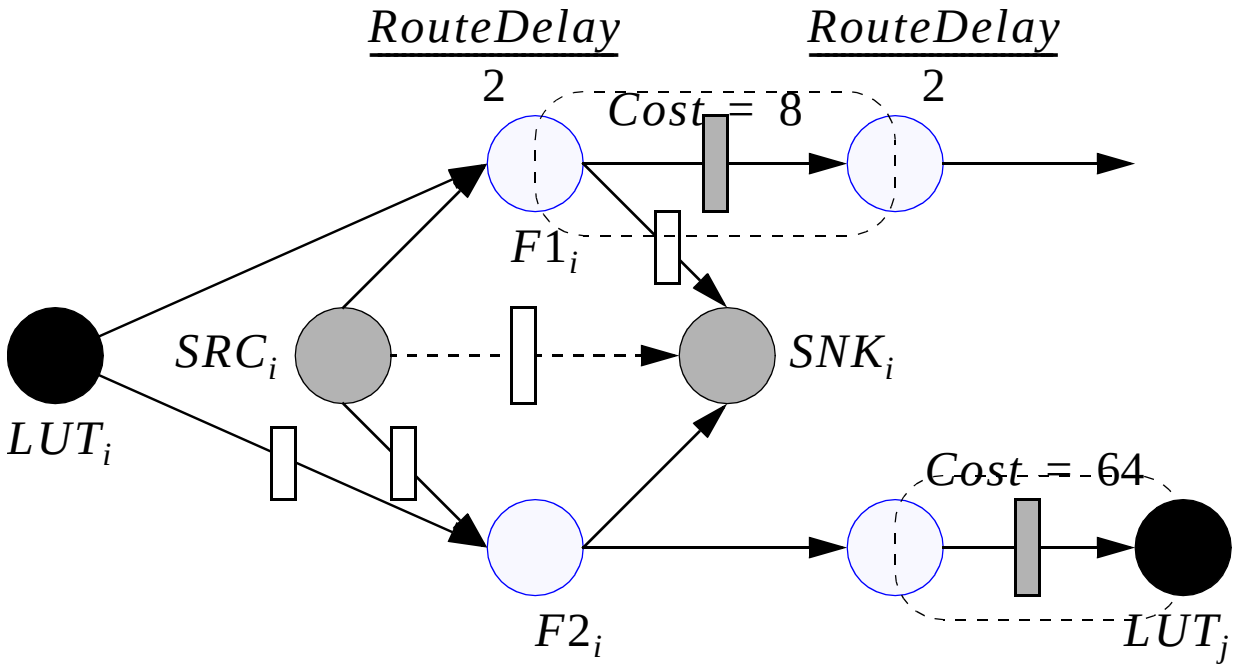

Figure 12: Costing Pipelined Routes and Fanin Registers.

then we must replicate the logic block. One block would be chosen to send out the combinational version of the signal, while the other would send out the registered version. Obviously it would be extremely beneficial to find a retiming solution that minimizes the amount of logic duplication needed to realize the target clock period.

Figure 11 shows our scheme for minimizing logic duplication during retiming. Figure 11(a) shows extra nodes added in the retiming graph to model the logic block. First an extra node is added on each fanout from the lookup table output. In the Figure, these nodes are labeled $F1_i$ and $F2_i$. A source node SRC_i and sink node SNK_i are added for each LUT. Wires are added from SRC_i to each fanout node, and from each fanout node to SNK_i. There is a virtual connection added from SRC_i to SNK_i. We now claim that minimizing the number of registers on the virtual wires minimizes the amount of logic duplication.

This claim can be explained fairly informally. Consider the situation shown in Figure 11(b). In this case we move only one of the registers forward from the fanout of LUT_i. There is now a single register on the virtual connection from SRC_i to SNK_i. Note that this is only possible by moving a register from the fanins of SRC_i. Since it has no fanins, this move is perfectly legal. There is no way of removing the register on the virtual connection, if the LUT provides both a combinational and registered output. In Figure 11(c), both registers are moved to the fanouts. Notice now that there is still a single register on the virtual connection. However it can be removed by pushing the registers on the fanins of SNK_i to its output. So minimizing the number of registers on the virtual connection again gives us the correct cost as no logic duplication is needed. Note that SRC and SNK vertices serve similar purposes to the *mirror* vertices described in [7] [8].

5.2 Costing Pipelined Routes and Fanin Registers

Figure 12 shows how we cost pipelined routes and fanin registers. It is occasionally very useful to pipeline a long route, to allow a signal multiple clock cycles to traverse a long stretch of interconnect. In a conventional architecture this is quite expensive. An ideal architecture might allow for the perfect division of routing delay by inserting a register. However inserting a register in the middle of a long route always has some overhead cost of going into the register and coming back out. In addition, there are reduced opportunities for register sharing amongst the various fanouts. Hence we try to avoid this situation as much as possible for both area and delay reasons. Thus placing a register in the middle of a connection is costed 8 times higher than logic duplication.

The figure also shows the situation where a register must be added at the fanin of LUT_j. This situation is the most expensive because each register on a fanin is assumed to be fairly close to LUT_j. However architectural constraints (discussed in the next section) imply that only a limited number of fanin registers can be placed physically close to their associated LUT. It would be difficult for the incremental clustering and placement algorithm to come up with a good solution for netlists with too many fanin registers. Thus, these fanin edges are costed at 64 times the cost of logic duplication.

The values discussed above were discovered through experimentation. However they are not very sensitive. Many different values work well. However the relative cost ordering of duplication, pipelining registers and fanin registers must be kept for effective retimings.

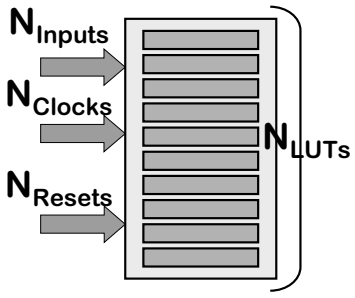

Figure 13: Simple Model of FPGA Cluster.

5.3 Solving the Minimum Cost Problem

The previous discussion established a modified retiming netlist and a cost for placing a register on certain edges. This problem is equivalent to minimizing $\Sigma_{e_{uv}} cost(w_{uv}) * w_{r,uv}$ while satisfying the constraints described in Equations 2, 3. This problem is the dual of the minimum cost network flow problem and can be solved via the Network Simplex algorithm. The techniques in [9] [14] are used to eliminate the redundant timing constraints defined by Eq. 3. Our implementation of the network simplex algorithm is extremely fast, as it takes less than 1 minute for even the largest circuits.

6. INCREMENTAL CLUSTERING AND PLACEMENT

The incremental clustering and placement algorithm (ICP) is used whenever the minimally placement disruptive retiming algorithm decides that it must add extra registers to the netlist. These extra registers must be placed at some location in the FPGA. Ideally these registers could be sifted into existing unutilized areas of the chip. However these retiming registers are added sparsely at random locations in the netlist, and placement algorithms tend to pack logic in tight clumps (little white space) to minimize delay and wirelength. Hence the ICP algorithm must try to create space for the newly inserted registers by moving the locations of non delay or wirelength-critical logic elements.

Moving logic elements in modern FPGAs is not necessarily a trivial process. Logic elements are typically organized into groups or clusters. Figure 13 shows a simplistic model of a FPGA cluster. Each cluster contains N_{LUTs} logic elements, N_{Input} input lines, N_{Clock} global clock inputs and N_{Reset} global asynchronous set/reset inputs. In academic literature [2], typical values for clusters are $N_{LUTs} = 4$, $N_{Inputs} = 10$ and $N_{Clock} = N_{Reset} = 1$. These constraints mean that even if there is a free space in the FPGA, we must check that there are enough input, clock, and reset lines available. For example a cluster using 4 LUTs but containing registers from two different clock domains is illegal.

The basic idea behind the ICP algorithm is to place these newly created registers (or duplicated logic) into their preferred locations even if it violates architectural constraints. For example if a particular LUT LUT_i must be duplicated, then the preferred location of LUT_i' is the same cluster as LUT_i. We then iterate on this starting point to try to remove the various architectural violations by moving non-critical logic.

The ICP algorithm is based upon an iterative improvement technique that moves logic cells in an attempt to minimize a cost function. This cost function includes the summation of three distinct parts:

- **Cluster Legality Cost** - Each cluster is penalized if it contains an illegal configuration. The cost is proportional to the amount of illegality.
- **Timing Cost** - The timing cost is used to ensure that critical regions of logic are not moved into places that would drastically increase the critical path delay.
- **Wirelength Cost** - Wirelength estimation is used to ensure that the circuit is easily routable after the logic element moves.

6.1 Cluster Legality Cost

There is a cluster legality cost associated with each cluster C_i. This cost can be calculated as shown in Eq. 7.

$$\begin{aligned} ClusterCost(C_i) = \quad & KL_i * overuse(C_i, N_{LUTs}) \quad + \\ & KI_i * overuse(C_i, N_{Input}) \quad + \\ & KR_i * overuse(C_i, N_{Reset}) \quad + \\ & KC_i * overuse(C_i, N_{Clock}) \end{aligned} \tag{7}$$

The notation $overuse(C_i, N_{LUTs})$ represents the number of extra LUTs contained in the cluster configuration C_i. The *overuse* function is defined similarly for input, clock and reset lines. The coefficients KL, KI, KR, and KC regulate the importance of the various types of overuse. These constants are all initialized to a value of 1, and gradually increased as shown in Section 6.5.

6.2 Timing Cost

One of component of the timing cost is based upon the cost used by the VPR placer. This cost is shown in Eq. 8.

$$Tcost_{VPR} = \Sigma_c crit(c) * delay(c) \tag{8}$$

This function encourages critical connections to reduce delay, while allowing non-critical connections are used to optimize wirelength and other optimization criteria. The ICP algorithm is not intended to improve the critical path delay of the circuit after retiming, but rather to preserve the delay by moving non-critical logic as little as possible. An aggressive cost function can cause non-critical connections to become critical. This is acceptable in a non-incremental placer because many moves can be made to correct this oscillation. However, ICP tries to make as few moves as possible because the retiming corresponds to the original placement. Hence we introduce a damping cost to prevent too many aggressive moves. It is shown in Eq. 9

$$\begin{aligned} Tcost_{DAMP} &= \Sigma_c max(delay(c) - maxdelay(c), 0.0) \\ maxdelay(c) &= delay(c) + \alpha * slack(c) \end{aligned} \tag{9}$$

The value of $maxdelay(c)$ is updated every time a timing analysis is executed. Its value is constant otherwise. The purpose of $maxdelay(c)$ is to control the delay expansion of a given connection. A delay that exceeds the maximum allocation is penalized while all other values are not costed.

6.3 Wirelength Cost

Figure 14 shows a high-level description of how the wirelength is monitored. Horizontal and Vertical cutlines are

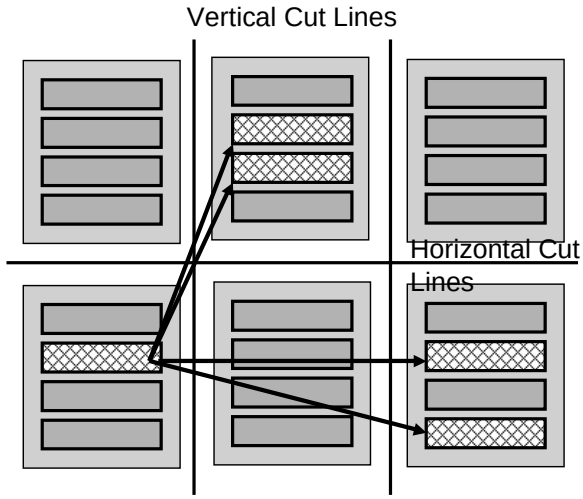

Figure 14: Local Congestion Estimation.

placed in each horizontal and vertical channel of the FPGA. The expected crossing count across each of these lines is monitored during each move. The average crossing count for every net can be computed using the techniques described in [3]. Monitoring the usage on these cutlines allows the measurement of localized congestion. In this way, the algorithm attempts to make sure that the new registers do not create any "hot-spots" that result in circuitous routes.

6.4 Move Proposals

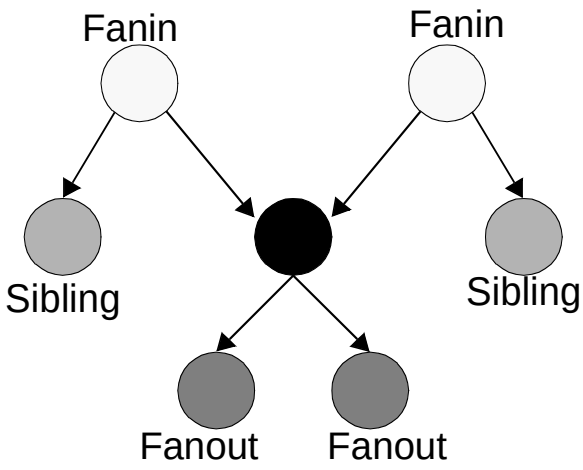

Figure 15: Fanin, Fanout and Sibling relationships.

Each iteration of the ICP algorithm chooses a candidate logic element LUT_i to move. Several different move types are selected in a random fashion. The various moves are:

- **Move-to-Fanin** - Attempt to move LUT_i to a cluster that contains a fanin of LUT_i.
- **Move-to-Fanout** - Attempt to move LUT_i to a cluster that contains a fanout of LUT_i.
- **Move-to-Sibling** - Figure 15 depicts the sibling relationship to LUT_i. Choose a sibling and attempt to move to the cluster that contains the sibling.
- **Move-to-Space** - Attempt a move to any random free slot in the FPGA.
- **Move in Direction of Critical Vector** - The critical vector for LUT_i is show in Figure 16. The direction of the critical vector is computed by summing the directions of all the critical connections attached to LUT_i. An attempt is made to move to a random cluster along the critical vector. This move helps to correct any mistakes when unexpected paths have become critical because of moves in previous iterations. Note that the critical vector move is similar to the move types attempted by iterative force directed placement algorithms.

Although move selection is random, the selected move is always biased in the direction of free slots.

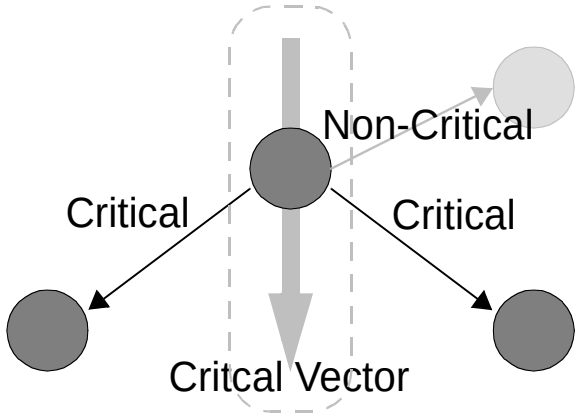

Figure 16: Critical vector.

Figure 17 shows a parent-move for a particular LUT. A register is inserted into a cluster causing it to violate architectural constraints as only 4 LUTs are allowed in each cluster. In this cluster, the LUT with the least timing critical connections is moved to a cluster containing one of its fanins. Although both of its fanin clusters has an extra free space, only one can possibly take this extra LUT since one of the clusters has already used up all of its input lines.

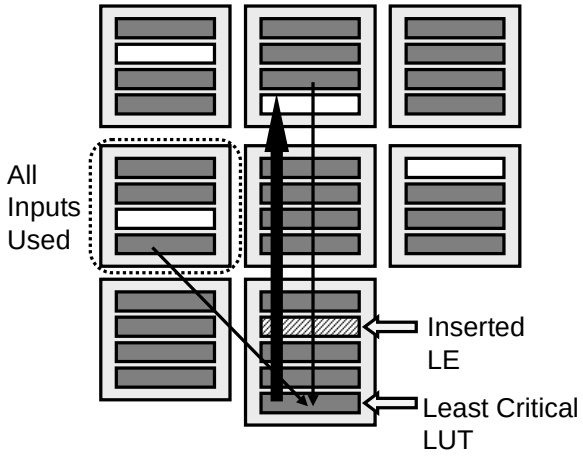

Figure 17: Example of ICP Move to Parent.

6.5 The ICP Algorithm

```
procICP
begin
    while there is overuse remaining
        choose any LUT_i from an overused cluster;
        select random move-type;
        evaluate change in cost ΔC;
        if ΔC < 0 then
          accept move;
        end if.
        every K iterations do
          run TA update crit(c) and maxdelay(c).
          call UpdateOveruseCoefs;
        end.
        if loopIterations > Threshold then
          return NO-FIT;
        end if.
    end loop.
end ICP.
```

Figure 18: Top Level ICP Algorithm

The basic cost function and move proposal schemes have been discussed above. Figure 18 presents the entire ICP algorithm. The algorithm simply chooses LUTs that participate in illegal clusters and tries to move them to improve the cost function. Notice also, that simple Timing Analysis (TA) is performed every K iterations. This call updates the *maxdelay* and connection criticality values to reflect the current configuration. The value of K is adaptively updated based on the amount of overuse remaining.

```
procUpdateOveruseCoefs
begin
    foreach overused cluster C_i do
      KL_i = KL_i + overuse(C_i, N_LUTs);
      KI_i = KI_i + overuse(C_i, N_Input);
      KR_i = KR_i + overuse(C_i, N_Reset);
      KC_i = KC_i + overuse(C_i, N_Clock);
    end loop.
end UpdateOveruseCoefs
```

Figure 19: Updating the Overuse Coefficients

Only moves that improve the cost function are accepted. Hence our algorithm is essentially greedy. The drawback with this approach is that the algorithm could easily get trapped in a configuration where it cannot find moves that decrease the current cost. To combat this problem, the **UpdateOveruseCoefs**, shown in Figure 19 is called every K iterations. It increases the overuse coefficients for every cluster that is illegal. This procedure actually reshapes the cost function to make it more favorable to move a logic element in an overused cluster since the overuse importance coefficients have been increased. This approach is similar to the Pathfinder [10] algorithm used for FPGA routing. However in this case LUTs "fight" for preferred cluster locations, by negotiating legality, timing and wirelength.

7. EXPERIMENTAL RESULTS

We conducted three experiments to compare different approaches. Our first approach is to retime the netlist at the LUT level (created by FlowMap [5]) and then perform the place and route steps. The retiming in this experiment is done with a unit delay model assuming that each LUT has unit delay. Our second experiment is to apply minimally placement disruptive retiming after the placement step. The placement engine uses conventional slacks rather than cycle slacks. The ICP algorithm is then run, and then the design is routed. Our third experiment goes through the entire flow proposed in this paper. Retiming-Aware placement is executed using cycle slacks. From this point the ICP and routing steps are executed. We show results on a cross section of circuits that we have studied in Table 7. Included are the largest of the sequential MCNC circuits and a few circuits gathered from free IP-core projects that were synthesized and mapped to simple LUTs and registers.

The target architecture for each circuit was selected so that the chip would be no more than 90% utilized. This number was chosen so that extra space would be available for the addition of retiming registers. In addition, a *low-stress* environment is assumed so that the router is given 20% more tracks than the absolute minimum required to route the circuit. The number of tracks is computed for the non-retimed circuit and not changed regardless of the number of registers that are inserted into the circuit.

The average improvement from integrating retiming with placement is approximately 19%, in comparison to retiming before placement, after going through the entire flow proposed in this paper. Notice that the cycle slack technique does not produce huge gains, but seems to consistently produce a slightly better clock period. This seems to be the case because most of the critical cycles in these circuits are close in delay to the critical path, so the unmodified VPR placer tries to keep them close together. The cycle slack technique seems to ask the placer to "try-harder" to keep the cycles close together. In addition to reducing the delay around the critical cycle, the delay of the near-critical cycles also seem to be reduced because there are fewer critical paths produced after retiming. This provides tools like the VPR router more freedom since it has fewer paths that must be routed on their preferred resources.

The minimally placement disruptive retiming algorithm has a run-time that is similar to conventional minimum-area retiming algorithms. However the actual execution time is strongly dependent on efficient implementation techniques [9] [14]. The execution-time of the incremental placement algorithm, is dominated by the timing analysis steps that take place every K iterations. However, if the timing graph is initially topologically sorted, then simple Timing Analysis can be accomplished in $O(n)$ time. Hence the ICP algorithm also runs in $O(n)$ time as the number of iterations attempted is bounded.

Table 1: Experimental Results

Circuit	Size (LUTs)	Retime First (ns)	ICP (ns)	Cycle Slack + ICP (ns)
bigkey-mcnc	1707	8.45	8.23	7.84
dsip-mcnc	1370	7.87	6.95	5.87
diffeq-mcnc	1497	15.65	14.83	15.01
elliptic-mcnc	3604	14.57	15.11	13.48
frisc-mcnc	3556	15.41	14.36	13.85
s38417-mcnc	6406	20.81	19.14	16.97
tseng-mcnc	1047	11.57	11.17	10.58
hc11-oc	3877	31.91	25.84	23.19
des-fip	15509	17.33	14.1	13.69
sisc8	1434	16.62	14.23	13.60
Average	4000	16	14.4 (+11.1%)	13.4 (+19.4%)

8. CONCLUSIONS

In this paper, we've shown that significant speed gains can be obtained by integrating retiming with placement algorithms for FPGAs in comparison to retiming at the LUT-level. The circuits that we've explored are fairly small in comparison to industrial standards. We feel that the small size actually hinders our experiments as larger circuits may experience larger routing delays in comparison to logic block delays because of the long distances that some paths must traverse.

9. 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] C. E. Cheng. RISA: Accurate and Efficient Placement Routability Modeling. In *ICCAD 1994*, pages 690-695, November 1994.

[4] B. Cherkassky and A. V. Goldberg. Negative cycle detection algorithms. Tech. Rep. tr-96-029, NEC Research Institute, Inc., March 1996.

[5] 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.

[6] J. Cong and S.K. Lim. Physical Planning with Retiming. Proc. *IEEE* International Conference on Computer Aided Design, pages 2–7, 2000.

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

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

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

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

[11] P. Pan. Continuous Retiming: Algorithms and Applications. International Conference on Computer Design, pages 116–121, 1997.

[12] P. Pan, A.K. Karandikar, and C.L. Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. Transactions on Computer-Aided Design, pages 489–498, 1998.

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

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

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