Retiming Tool

CSEE E6861y - Computer-Aided Design of Digital Systems

Kshitij Bhardwaj kb2673@columbia.edu
Paolo Mantovani pm2613@columbia.edu

ABSTRACT

In the context of logic synthesis, the retiming step is performed on a multi level logic network, in order to rebalance combinational paths and to squeeze some slack to increase clock frequency. The tool takes as input a Control Data Flow Graph (CDFG) encoded as a Synchronous Logic Network Graph, which will be addressed simply as graph or network through the rest of this report.

The tool has two working modes: `min cycle` and `min area`. The former mode aims to find the minimum feasible clock cycle and retimes the network regardless of the registers count. The latter, instead, takes as input a target feasible clock cycle and tries to retime the network so that the clock cycle is met and the number of registers is minimized.

1. TOOL OVERVIEW

The program is written in `C++`, which compiles to fast native executable and the code consists of a main function, which calls the basic components of the program:

- **The parser**: takes one text input file which describes the graph, by listing nodes delay (combinationa logic) and edges weight (registers current position). The parser allows the presence of comments at the end of each line.
- **Min Cycle**: is the defaul running mode, which relies on the following basic algorithms, which are invoked by the main function:
  1. **WD**: computes all paths delays and weights for each vertex pair and stores them into two matrices (W and D). This steps is an adapted version of `Floyd Warshall` algorithm.
  2. **Binary search**: pick values from D matrix as possible target clock cycles.
  3. **FEAS**: for each feasible target cycle, this step returns the retiming vector and the retimed network. The algorithm repeats the following steps n times, where n is the number of nodes in the network.
     - **Clock Period (CP)**: computes the arrival time of each vertex, given the current register placement.
     - **Retime**: determine the next retiming vector, based on late notes, retime then retime the network.
- **Printer 1**: is responsible for invoking the methods for printing the initial network information, the retiming information and the minimium cycle achieved. When `verbose` flag is enabled, all intermediate steps of WD and FEAS are printed to file.
- **Min Area**: takes the same input file described above, plus the target clock cycle. This mode performs the following basic steps:
  1. **WD**: same as above.
  2. **SIMPLEX**: given the information found in the previous step, the retiming problem is expressed as a set of linear disequations and a function to minimize, which is the total number of registers after retiming. The disequations are the constraints to which the network is subject to: there must be no combinational loops and the final circuit has to meet the target clock cycle. SIMPLEX class includes, among the others, the following important methods:
     - **Make Tableau**: transforms the constraints and the objective function into a matrix.
     - **Phase 1**: performs rows operation to get a feasible solution to the linear system.
     - **Phase 2**: searches for the legal and feasible retiming vector that minimizes the registers count. This is done again by performing row operations on the matrix or `tableau`.
- **Printer 2**: is responsible to print the initial network information, the retiming information and the minimum area achieved. When `verbose` flag is enabled, all intermediate steps of SIMPLEX are saved to file.

Further details on the algorithms are provided in sections 2 and 5.

2. DATA STRUCTURES AND ALGORITHMS

This section describes in details the data structures used for each part of the and gives some insights on the way the main algorithms have been coded.

2.1 Storing the synchronous network

To store the graph we designed a class which can serve both modes of operation of the tool and allows for a clean and readable programming style. Also through a massive usage

of pointers, we avoid having multiple copies of the network, thus reducing the memory requirements. This class is named `sng` and a part from a string, which stores the name of the circuit, it contains two main member variables: a vector of pointers to edge and a vector of pointers to vertex. Edge and Vertex are the name of two classes, which keep all the information necessary to run the algorithms of the tool and to print intermediate and final results.

Vertex includes the following items:

- **in**: is a vector of pointers to edge that contains a pointer per each edge connected and directed to the vertex.
- **out**: is a vector of pointers to edge that contains a pointer per each edge departing from the vertex.
- **id**: is the name of the node assigned based on the input file.
- **delay**: is the combinnational delay of the node.
- **delta**: is the arrival time, including the delay of the node, which is set by CP to determine the current clock cycle and which nodes are late.
- **color**: is a flag used when walking the tree to perform Depth First Search (DFS) on the graph.
- **c**: is the difference between the vertex in-degree and out-degree, which is used to generate the objective function to minimize for the `min area` mode.

Edge includes the following items:

- **id**: is the name of the node, assigned based on the order in which vertices are listed in the input file. Notice that the structure of the produced graph does not depend on the edge ordering, however the id of the edges does.
- **weight**: is the number of registers present on the edge.
- **src**: is a pointer to the source vertex of the directed edge.
- **dst**: is a pointer to the destination vertex of the directed edge.

Among the methods available in class `sng`, it is worth to mention:

- **reset_deltas**: this method resets the arrival time of the vertices, which is necessary before running CP after the network has been retimed.
- **retime_sng**: this method takes as input a retiming vector and applies it to the graph. All edges weigths are properly updated.
- **revert_sng**: this method takes as input a retiming vector and reverts the network retiming. FEAS calls this step when a target clock cycle is non feasible. Also we revert to the graph every time we try a new target cycle. Being able to revert a retiming, allows us to avoid copying the graph while running the tool.

2.2 WD calculation

Matrices of the arrival times D and of the paths weights W contains a fixed number of elements, which is the square of the number of vertices. Therefore we choose to avoid using dynimic memory allocation ad to represent the two matrices as two dual dimension arrays of size n by n.

The class dedicated to WD algorithm has also a vector of unsigned integers, which is used to store and sort all possible values to be picked as target clock cycle in `min cycle` mode. The values stored in this vector are dumped from matrix D, then we sort the vector and remove all duplicates.

The methods of this class are the basic steps of the Floyd-Warshall algorithm:

- **initialize WD**: we initialize the matrix based on the `sng` graph. At the beginning, we only know verteces delay and edges weight. All paths not yet discovered are marked with an infinite weight and 0 delay; on the other hand, all paths corresponding to an edged are marked with the edge weight and the delay of the source vertex.
- **compute WD**: we run a single pass of Floyd-Warshall and we update while the algorithm runs both paths weight and delay, thus we find per each path the `min-weight - max-delay`. During the last iteration of the outer loop of Floyd-Warshall we add to the paths the delay of the destination vertex and we determine the current minimum clock cycle of the network.

2.3 FEAS algorithm

The class implemented for FEAS does not need any additional data structure. It includes: a copy of the pointer to the network of type `sng`; a class, named `cp`, which includes the methods to run DFS on the graph and determine the feasible clock cycle after each retiming step; a set of auxiliary methods to run the kernel of FEAS.

To avoid copying the network, at each iteration of FEAS, we retime the network starting from the graph obtained at the previous iteration. We also update the elements of the retiming vector, with respect to the input network, but we use a local retiming vector at each iteration, which represents the retiming between iteration i and iteration $i-1$. Finally, if the retiming was successful, we return the global retiming vector along with the retimed graph, otherwise, we revert the network before retunring to main.

2.4 Linear programming: constraints generation

While the tool is able to generate the tableau to run SIMPLEX in a single step, starting from matrices WD and from the input graph, we added additional methods to generate and sort the list of constraints to which our linear program is subject to.

At first we add to the list of constraints the necessary conditions to get a legal network, which implies that no edge has a negative weight. Per each constraint added, we introduce

a new slack variable to the system and we add a row to the matrix.

As a second step, we add the timing constraints, derived from WD matrices. This time, since from the matrices we get some redundant constraints, we mark the required ones, so that we can print both a complete list of constraints and an optimized one. However, we only add to the matrix the reduced list of constraints, while we skip the others.

To support sorting and printing we use a vector of type `p2_triple`, that includes the id of the vertices involved, the coefficients of the disequation, the sign of the answer term and a flag to distinguish between redundant and necessary constraints.

2.5 Linear programming: SIMPLEX

SIMPLEX algorithm relies on the representation of the tableay, which is a vector of rows, where each row is a class that contains the following items:

- **label**: the variable for which the row tells the basic solution.
- **coeff**: a vector of integers, which are the oredered coefficients of the disequation represented by the row. The first n coefficients are related to the retiming vectors elements (one per each vertex of the graph), while the subsequen numbers relate to the slack variables used for SIMPLEX.
- **ans**: is the answer term of the row.
- **ratio**: is a string updated at each iteration of FEAS, while searching for the right pivot in the matrix. This string is useful only in verbose mode to print intermediate results, while the actual ratio is not stored, because we determine the pivot with a single pass across all the rows.
- **star**: is a flag that tells if the coefficient of the variable represented by the label of the row is negative. This flag is used in phase 1 of FEAS to manipulate the matrix in order to obtain a feasible solution for the linear program.

The class `simplex` contains a collection of methods to create the tableau, by adding all the rows derived from the constraints. Moreover, the last row added to the matrix corresponds to the objective function, determined based on the information stored into the graph, and in particular, the `c` value of each vertex.

Most important methods called in `simplex` are functions to find the appropriate pivot columb, to determine the pivot row and to clear the column of the pivot with a simple row operation applied to the entire matrix.

It is worth to mention that in verbose mode only, the rows in the tableau are ordered before starting FEAS kernel, so that each row of the initial tableau corresponds to the disequation in the reduced set of constraints listed in the same order.

3. TESTING METHODOLOGY

Due to time shortage, we decided to design and test our program incrementally, running small simple intermediate tests to check the correctness of every algorithm or portions of them.

We produced some hand solvable inputs for Floyd Warshall, CP, FEAS, network retiming and reverting, constraints generation step, objective function generation, SIMPLEX and we ran all the tests separately while coding the related algorithm. Notice that SIMPLEX was also tested agains a web based applet that solves linear programming using SIMPLEX method.

Thanks to the flexible data structures and the methods interface we provide, we could reliably assembling the tool, starting from tested basic components, with a low probability of introducing new bugs.

We also used compiler pragmas to enable two levels of degbug:

- INFO: when defined we print all basic information of the algorithms, along with some intermediate results. This level of information is easy to read and allows to quickly identify at which step of the computation the error occured.
- DEBUG: when defined we print detailed information of every function that makes some computation. The tool prints a huge amount of extra information, which most of the time allows to identify the cause of the error. It is recommended to get first the basic information using the definition INFO and finally to use the DEBUG information to dig into the problem

To test the entire tool in both modes we leveraged the two given examples of the correlator. Also we tested the tool with a larger network, which was not strongly connected. This test helped us to check if the tool behaves correctly when WD matrices are incomplete. All the three examples were compared against a manual solution of the retiming problem.

Most of the issues we faced were cause by a slighlty incorrect formulation and set up of the problems, rather than by the implementation of the algorithms, which are small and simple to code.

4. ADDITIONAL INSIGHTS

In section 2 we described in detail our choices in terms of data structures that makes our retiming tool fast and memory efficient.

It is worth to mention a few more features of the tool, which we implemented:

- Reduced number of loops, obtained by computing all results on-the-fly, rather than iterating many times to produce incrementally the information. The drawback of this optimization is that we had to step back and

massage the code, in order to print all the required outputs, which are sometimes irrelevant to the computation of the final result.

- Each vertex and each edge is "self contained": no need for iterating over the list of edges or over the list of vertices to retrieve the information needed, because each vertex has a pointer to each edge which impinges it. Similarly each edge has pointers to source and destination vertices.

- Binary search for `min cycle` prunes the search to values which are smaller than the initial clock cycle.

- Tableau rows labeling for `min area` uses numbers instead of strings and a member variable keeps track of each added row and slack variable, thus making easier to perform row operations and to read the basic solution from the tableau.

5. TECHNICAL ISSUES

5.1 Floyd Warshall Algorithm

1. Iteration 0 ($k = 0$): all paths not yet discovered are marked with an infinite weight (Matrix W) and 0 delay (Matrix D); on the other hand, all paths corresponding to an edged are marked with the edge weight and the delay of the source vertex. Also, edge weights between the same nodes is set to 0.

2. Iteration 1 ($k = 1$): At the end of first iteration, matrix W has the following entries $wuv(1)$: $wuv(1) = min(wuv(0), wu1(0) + w1v(0))$ where $wuv(0)$ is the minimum weight between u and v vertices from iteration 0, $wu1(0) + w1v(0)$ is the minimum weight between vertex u and vertex v, if vertex 1 is an intermediate vertex of the path between u and v. The second expression in the min function is again calculated from the minimum distances computed during iteration 0. Any entry updated in matrix W causes an update of the same entry in matrix D, which keeps track of the total delay between the two vertices uv, when the path with weight wuv(1) is selected.

3. Iteration i ($k = i$): At the end of ith iteration, matrix W stores the minimum weight between any two nodes u, v, if there exists a path from u to v of length at most i. Matrix D stores the total delay of such a path from u to v. Each element is calculated with the same expression shown for step 1, in which we have i in places of index 1 and $i - 1$ in place of index 0.

5.2 Retiming: Handling too slow operators

The timing constraints for the retiming problem state that

$$\forall V_i, V_j s.t. D(V_i, V_j) > \phi \qquad R_i - R_j \leq W(V_i, V_j) - 1 \quad (1)$$

If we have $D(V_k, V_k)(= 30) > \phi(= 20)$, then the following equation must be satisfied:

$$R_k - R_k \leq W(V_k, V_k) - 1 \qquad W(V_k, V_k) = 0 \quad (2)$$

The above inequality will clearly never hold and therefore the system of equations will always be unsolvable.

5.3 Retiming: Exploring alternative retiming vectors

1. Legal retiming vectors r_x, larger than r_{opt} are not always feasible. The following retiming vector gives a counter example: -[1 2 2 3 1 1 0 0]. Retiming value of vertex V_e in figure 9.8 (DM) is increased from -2 to -1. Equation 9.4 is still satisfied, which implies a legal vector, but equation 9.5 is not satisfied for vertices V_e and V_f.

$$R_e - R_f \leq W(V_e, V_f) - 1 \quad (3)$$

   Computing the left hand side (LHS) of the above equation we have $-1 - (-1) = 0$ which is not less than or equal to -1. Hence the retiming vector is infeasible.

2. Similarely we provide a counter example for smaller retiming vectors. -[1 2 2 3 2 2 0 0]. Here retiming value of vertex V_f is decreased from -1 to -2. Equation 9.4 is still satisfied, indicating a legal vector, but equation 9.5 is not satisfied:

$$R_e - R_f \leq W(V_e, V_f) - 1 \quad (4)$$

   LHS: $-2 - (-2) = 0$, which is not less than or equal to -1.

3. None of the smaller legal retiming vectors are feasible for the correlator example. In the five relevant inequalities corresponding to equation 9.5, if we reduce the retiming values of the vertices that are used in these inequalities then we either get a retiming vector which is not legal or it does not satisfy the feasibility constraint. Looking at the variables R_i and R_j (corresponding to vertices in the five inequalities) in equations 9.4 and 9.5. If we reduce the value of R_i then we either violate the legality constraint (for vertex 1) or violate the feasibility constraint. On the other hand if we reduce the value of R_j then we will always violate the feasibility constraint.

5.4 SIMPLEX: selecting a pivot columb

1. Choice of negative column: The goal of simplex algorithm in each iteration is to reformulate the linear program so that the basic solution has a greater cost (for a maximization problem). Following this principle, the pivot column is selected, contains a negative number in the bottom row as the bottom row represents the objective function and the negative number is actually a positive coefficient of a non-basic variable in the original objective function. Increasing the value of this non-basic variable will lead to an improved cost during each iteration.

2. Other negative columns: **Iteration i vs $i-1$**: The cost of the current iteration will be higher as compared to $i - 1$. This is because a non-basic variable is selected which has a positive coefficient in the original objective function. **Iteration i (alternative negative pivots)**: The objective cost for the current iteration will be higher if we use greatest magnitude negative pivot column. This is because the greatest magnitude negative column corresponds to the greatest positive coefficient of original objective function.