CHAPTER 3: DESIGN AND IMPLEMENTATION OF COORDINATION TECHNIQUES FOR CCHBS

The maximum usage of FPGA is achieved only when using appropriate methods of programming, properly coordinating the configurations and interfacing from PCs. This work deals with the precise technique of configuring multiple independent CCHBs of the applications in FPGA. In this chapter, the various methods involved for the development of FPGA accelerated system is discussed. First step is, investigates of proper identification of all the potential CCHBs from application systems. The identified CCHBs are then subjected to a heuristic method called conflict graph approach in order to select the best among all possible CCHBs. The best CCHBs are merged together with the use of multiplexers by checking the number of common operations among them. The reconfiguration of CCHBs is needed only when the identified CCHB set could not be accommodated into the FPGA at single instance of time and when dependency exists among them. The circuits of the scheduled CCHBs are configured into the FPGA when scheduling is done.

3.1 OVERVIEW

The computation intensive parts of the applications are implemented as CCHBs. CCHBs are identified by profiling the entire DFG of the application for their dynamic occurrences in the application using Trimaran compiler [26]. Sub-graph isomorphism method in the vflib graph-matching library is used to find all the potential CCHBs from the DFG [27]. Memory operations are not included in selecting CCHBs. The arrow from
each application indicates that it is subjected to modification of C Code after identifying CCHBs. The applications are assumed to execute in random order.

The independent CCHBs are determined by identifying maximum independent set of the independency graph. The CCHBs in maximum independent set are removed separately and the process is repeated to find out the next set of independent CCHBs. Then, two or more independent CCHBs that can be configured simultaneously are combined using data-path-merging. Data-path-merging is performed by determining the common operations among the CCHBs. Multiplexers are inserted appropriately after merging the common nodes of CCHBs. The merging results are verified by determining the cycles in the data flow graph after merging. The merge which causes additional cycles is then removed. The merging process continues until the area is maximally utilized.

Figure 3.1 The architecture framework.
The complete framework of scheduling and configuring the CCHBs on FPGA is depicted in figure 3.1. The scheduler is to schedule the CCHBs in an efficient way to improve the utilization of the hardware and reduce the configuration bits when reconfiguration is required for executing dependent CCHBs in successive configurations. Based on the attributes of CCHBs such as CCHB size, speedup, and CPU Cycles needed if executed in Software (CCS) and the FPGA size, the gain factor of each CCHB is determined. The CCHBs are arranged in decreasing order of their benefit. The hardware definition language (HDL) code for those CCHBs are implemented and synthesized into bit files. The configuration bits of the CCHBs are downloaded and configured into the FPGA. The scheduling is performed in this way to accelerate many different application systems or distinct calculations within the same application system.

3.2. TEMPLATE LIBRARY

The template library is the collection of templates that are pre-generated and stored in database [82]. The templates in the template library are collected from a set of applications belonging to various application domains. Those templates are the frequently occurring ones in the applications of various domains. The templates in the template library may be of different size. The smaller one may be the subset of larger templates.

3.3 IDENTIFICATION OF CCHBS

The CCHBs are the instructions that are executed by the special purpose hardware. The CCHB implements the computation intensive part of the application. Both the templates and the application are in the graphical representation. The number of
matches of templates in each and every basic blocks of an application for all templates in the template library is determined using VFlib graph matching library. VFlib deals with simple directed graphs which cannot contain self loops, the edge (i, j) is considered different from edge (j, i) and between two nodes there is utmost one edge. The CCHBs in the application are identified by tracing the DFG of the application. Using the graph - sub graph isomorphism, the nodes of DFG are compared with the templates that exist in the template library, and then all possible matches are determined. The identified matches are CCHBs which are processed by the conflict graph algorithm to select the best CCHBs.

![Diagram of Template1 matches](image1)

Figure 3.2 Matches belonging to Template1 of template library within the same basic block.

![Diagram of Template2 matches](image2)

Figure 3.3 Matches belonging to Template2 of template library within the same basic block.
The matches belonging to the same template and the same basic block are linked together under the template in the template link library. Then the matches belonging to the same template and different basic blocks of the application are found and collected as a group of matches. The matches which are from the same basic block may contain same nodes (i.e. overlapping of nodes between matches of same basic blocks are allowed) as shown in figure 3.2 and 3.3.

There exist three and two matches for the templates - Template1 and Template2 in one basic block of the application. In that, the matches 1 and 4 are overlapping in which the node n6 is common. Similarly the matches 2 and 5 are overlapping in which node n21 is common.

3.4 SELECTION OF CCHBS

The CCHBs that are identified from the previous section are then processed in order to select best CCHBs. The gain factor of CCHBs in their corresponding basic blocks is also taken into consideration during selection. It will be more useful if the selected CCHBs mostly belong to same template rather than to different templates. Hence grouping of CCHBs belonging to the same template is needed and number of CCHBs in one template also accounts for the selection process. If more number of CCHBs are present in one template and their gain factor is also high compared to others, then those CCHBs will be having higher probability of selection. In order to select best CCHBs, a heuristic approach is used. The conflict graph algorithm [34] is slightly modified to meet the above requirements.
3.4.1 CONFLICT GRAPH

A conflict graph is an undirected graph $G (V, E)$, where $V$ is the set of CCHBs of any template and $E$ is the set of edges between vertices to signify that the CCHBs have common operations represented as nodes of DFG. The identified matches from the application are examined carefully to find out the subset matches. The subset matches may sometimes reduce the opportunity of their superset matches to be selected as best CCHBs. So at the beginning of the algorithm, those subset matches are identified and discarded.

The following is the pseudo code for identifying and discarding the subset matches.

Repeat for each match $i = 1$ to $n$ that are identified from the application

Repeat for each match $j = 1$ to $n$ that are identified from the application

If $i \neq j$ then

If $match_i \subseteq match_j$ then

Discard $match_i$;

endif;

endif;

end for;

end for;
The matches of Template1 and Template3 are shown as example in figure 3.4. In that, the match6 is the subset of match1, match7 is the subset of match2 and match8 is the subset of match3. so the matches of Template3 in basic block1 are discarded in order to improve the opportunity of selection of superset matches - match1, match2 and match3 of template1 as the best CCHBs to increase the performance.

The conflict graph is constructed by determining the edges which is possible between the CCHBs, if they contain overlapping nodes (i.e. common node present in two or more CCHBs). The matches possessed by the same template are grouped together as shown in figure 3.5. The following pseudo code is for grouping the matches under their corresponding templates.
Initialize $\text{template\_ptr} = \text{NULL}$;

Repeat for each match $i = 1$ to $n$ that are determined in the application.

    Search for the $\text{template\_id}$ in the $\text{template\_ptr}$;
    
    If the $\text{template\_id}$ is in $\text{template\_ptr}$ then
    
        Add $\text{match}_i$ under the $\text{template\_id}$;
    
    Else
    
        Add new $\text{template\_id}$ of $\text{match}_i$ into $\text{template\_ptr}$;
        Insert $\text{match}_i$ into just added $\text{template\_id}$;
    
    endif;

end for;

The $\text{template\_ptr}$ is initially assigned as NULL. The $\text{template\_id}$ of the first match is added into the $\text{template\_ptr}$. The first match is then added under its $\text{template\_id}$. The $\text{template\_id}$ of the second match is examined for its presence in the $\text{template\_ptr}$. If it is present, the second match is added under it. Otherwise the new $\text{template\_id}$ is added into the $\text{template\_ptr}$ and the match is added under it. This process is repeated for all the selected matches. Finally the matches are grouped under their templates and the set of templates with their matches forms the conflict graph.
3.4.2 MAXIMUM INDEPENDENT SET

In each and every group of templates the maximum independent set (MIS) is found. Finding the maximum independent set is again heuristic. The resultant maximum independent sets of each template group are then sorted according to their weights. The weight can be adjusted corresponding to the application dealt with. Generally the weights are the number of nodes in the maximum independent set. In this work, the gain factor of the CCHBs is also included in the weight. The procedure of finding the maximum independent set is as follows. Initially the maximum independent set will be empty. A node with minimum degree in the each template is found. If there is more than one node, then a tree should be constructed.

3.4.3 TREE STRUCTURE

The tree is constructed in such a way that selection of a node leads to the deletion of its neighboring nodes and in turn gives the possibility of selecting the neighbors of its
neighboring nodes. The root of the path with highest weight is selected as the node with minimum degree as shown in figure 3.6.

![Figure 3.6 Tree structure that resolves the conflict of minimum degree node while finding the MIS.](image.png)

The node with the minimum degree is included in the maximum independent set. Then the node and its neighbors are deleted from the template which is considered to find the maximum independent set. Then again, the next node with minimum degree is selected from the remaining nodes of the template and this procedure is repeated until the template is exhausted. The collection of nodes with minimum degrees is then the resulting maximum independent set of that template. Maximum independent set for every template is determined in the same way.

The selected maximum independent CCHB sets from each template are sorted according to their weight as illustrated in figure 3.7. The maximum independent set with maximum weight is then chosen. The CCHBs which are overlapping with the selected maximum independent set CCHBs are discarded. This discard should be performed for the entire graph and not be restricted to that particular template.
There exist two steps of discard of CCHBs. First, when finding the maximum independent set for each template, the neighboring CCHBs within the template are discarded. Hence it is termed as local deletion. Second, after choosing maximum independent set with maximum weight, the neighboring CCHBs of the entire MIS is discarded. Hence it is termed as global deletion.
The maximum independent set of template T1 has the higher weight, it is selected and the global deletion takes place. After regaining all the nodes that are discarded in the local deletion except the nodes discarded in the global deletion, the graph is subjected to the second iteration and the process is illustrated in figure 3.8. The same process is repeated until the graph is exhausted. According to the example discussed, three iterations are possible and the third iteration is demonstrated in figure 3.9.

3.4.4 CONFLICT GRAPH ALGORITHM

The conflict graph algorithm is implemented by considering the edge information in a matrix that has the dimension n x n (where n is the no. of matches in the template) for each template. Also another matrix is constructed for the entire conflict graph. The entries in those matrices are only 0 and 1. If match i and match j overlap, then the entry (i
* n + j) and (j * n + i) should be set to 1, otherwise 0. Figure 3.10 shows a sample matrix. Obviously, those matrices are symmetric matrices. The following pseudo code creates those matrices and fills its entries.

Repeat for each template i = 1 to t in the conflict graph

    Repeat for each match j = 1 to n

        Repeat for each match k = 1 to n

            If match j and match k are overlapping then

                Set the template_mtx; at (j * n + k) and (k * n + j);

            endif;

        end for;

    end for;

end for;

similarly,

Set the conflictgraph_mtx of the conflict graph accordingly if the matches with in the entire graph are overlapping;

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0
\end{pmatrix}
\]

Figure 3.10 Matrix structure to maintain the edge information.

The MIS should be determined initially, in each template. There exists a procedure “MIS(current_template)” for finding the MIS. After finding the MIS for each template, the MIS with maximum weight/gain is selected. The matches in the selected
MIS are added into the *best_matches*. The neighboring matches of the MIS are then deleted from the entire graph. Their entries are reset to 0 in the template matrices and in the entire conflict graph matrix accordingly. The template from which the MIS is selected should not be considered in further iterations. The above procedure should be repeated until the conflict graph becomes empty. The *best_matches* finally holds the set of best CCHBs. The gain of the MIS is as follows.

**Weight of MIS of \( T_j \) = \( n \times \text{no. of nodes of the template} \times (\text{no. of nodes of the template})^{0.2} \times \sum_{i=1}^{n} \text{dynamic\_occurence}_i \)**

Where \( n \) = no. of matches in the MIS of the template \( T_j \)

The term \((\text{no. of nodes of the template})^{0.2}\) is needed to give priority to the larger size templates.

The following pseudo code is the actual algorithm that selects the best matches from the conflict graph.

```plaintext
best_matches = NULL;
Repeat until the conflict graph is exhausted
Repeat for each template \( i = 1 \) to \( t \) in the conflict graph
   \( mis_i \) = MIS(template\(_i\));
end for;
calculate the gain of each MIS;
select the MIS with maximum gain;
add it into the set of *best_matches*;
Delete the matches which are in the MIS and their neighbors in the conflict graph;
end for;
```
procedure MIS(current_template)
begin
mis = NULL;
while current_template ≠ ε do
    choose match v such that degree(v) is minimum;
    mis = mis U { v };
    current_template = current_template - { v } U neighbors(v);
end while;
return mis;
end proc;

The MIS can be determined by searching the minimum degree node/match within the template. While determining MIS, matches are the nodes. The degree of the node signifies the no. of overlapping of the matches within the template. If more than one minimum degree node is there, then the conflict in selecting between the two nodes is resolved by constructing the tree structure. The tree structure has the dummy root node which in turn has the children as the conflict nodes. The tree is grown by considering its neighbors as its children. The leaf node is recognized by identifying all its neighbors which are already present in the tree structure. The gain for each node is computed as follows and finally from the result, the minimum degree node is selected.

The weight of each node(CCHB) in the tree structure is calculated as follows:

node_weight = dynamic_occurence * (percent_speedup/100);

where dynamic_occurrence is the no. of occurrence(s) of the template in the application,
percent_speedup is the percentage of speed of the template.

After finding the minimum degree node, its neighbors within the template should be deleted and the minimum degree node just determined is added into the corresponding MIS of the template. Then with the remaining set of matches/nodes, the entire process is repeated.

3.4.5 RESULTANT CCHBS

After selecting the best CCHBs from the identified set of matches, they are sorted according to their execution order in the application. The number of CCHBs in the sorted CCHB list may be more than the actual number of CCHBs selected. The reason for this is one CCHB may be executed more than once. This may occur if the function or a basic block in which the CCHB is identified, is executed more than once.

3.5 DATAPATH MERGING

Each piece of the application is represented as a DFG. The DFG merging process identifies similarities among the DFGs, and produces a single data path that can be dynamically reconfigured and has a minimum area cost, when considering both hardware blocks and interconnections.

![Data flow graph merging](image)

Figure 3.11 Data flow graph merging.
Figure 3.11 illustrates the concept of data-flow graph merging. When DFGs G1 and G2 are merged, the resulting data path is produced. In the resulting data path there are edges originated from only one DFG (e.g., the (+, x) edge from DFG G1) and edges shared by both DFGs (e.g., (+, -) edge).

Two successive CCHBs are merged in order to reduce the area they occupy if they were considered as individual CCHBs. So merging of CCHBs will yield a considerable reduction in the hardware area. If the successive CCHBs are more similar, then higher is the benefit gained in terms of reduction in area. Obviously the cost of multiplexers is also included while merging. If the two nodes are merged then checking of their inputs is mandatory for the inclusion of multiplexers.

After including the multiplexers, if the area still remains, then merging of next CCHB is processed and checked with the area and the process continues till the area cannot accommodate the next CCHB including of multiplexer. In this work, the data path merging is implemented using a heuristic algorithm. The compatibility graph algorithm is slightly modified to work with the above problem.

### 3.5.1 COMPATIBILITY GRAPH ALGORITHM

The given two graphs are analyzed to find the similarity between them.

1. Find the common nodes and edges and map them by considering all the possible combination.
2. Construct the Compatibility graph.
3. Find the Maximum clique in the compatibility graph. (If more than one exists, choose any one).

4. Build the merged graph with the nodes and edges of maximum clique obtained in the above step.

5. Include the rest of the nodes and edges which are not in the maximum clique, from graph1 and graph2 into the merged graph.

6. Include multiplexers appropriately.

### 3.5.2 STEP BY STEP GENERATION OF MERGED GRAPH

Initially, a compatibility graph is constructed to represent all possible vertex and edge mappings between two input DFGs G1 and G2 and the consistency among these mappings. Two vertices from G1 and G2 respectively, can be mapped (overlapped) if those are in the instruction set and performing the operations they represent are same. Two arcs from G1 and G2 can be mapped if their corresponding source vertices can be mapped, as well as their destination vertices.

Two vertex or edge mappings are not compatible whenever they map the same vertex of G1 to two different vertices of G2, or vice-versa. This compatibility criterion is illustrated in figure3.12. In that figure two DFGs G1 and G2 are shown. There are two possible arc mappings between G1 and G2, which are \((a_1, a_3/b_1, b_4)\) and \((a_2, a_3/b_3, b_5)\). These two mappings are not compatible since they map the same vertex \(a_3\) from G1 to two different vertices, \(b_4\) and \(b_5\), in G2.
3.5.2.1 NODE FORMATION OF COMPATIBILITY GRAPH

The following pseudo code is for the creation of compatibility graph. The graphs G1 and G2 are the DFG representations of the given CCHBs. The operation of each vertex of DFG G1 is taken and match for that operation is searched in the operations of each vertex of DFG of G2. If any two operations are matched, those two vertex mappings are added into the compatibility graph as a node.

The following pseudo code is to perform the vertex mapping.

Let \( G_1 (V_1, E_1) \) be the DFG of C1 and \( G_2 (V_2, E_2) \) be the DFG of C2;

Repeat for each vertex \( u \) in \( V_1 \) of \( G_1 \),

Repeat for each vertex \( v \) in \( V_2 \) of \( G_2 \),

If \( (u.operation = = v.operation) \) then

Add \( (u, v) \) vertex pair as a node into compatibility graph;

endif;

end for;

end for;

Edge mappings are performed, similar to the vertex mapping. In the case of edges, the search for matching of edges in the two graphs G1 and G2 involves two conditions.
The two edges are said to be matching, if and only if the source vertex from graph $G_1$ and source vertex from graph $G_2$ are mapped and the destination vertex from graph $G_1$ and destination vertex from graph $G_2$ are mapped.

The following pseudo code is for edge mapping:

Repeat for each edge $p\rightarrow q$ in $E_1$ of $G_1$,
Repeat for each edge $r\rightarrow s$ in $E_2$ of $G_2$,
If ($p.operation = r.operation$) and ($q.operation = s.operation$) then
Add ($p, q, r, s$) edge pair as a node into compatibility graph;
endif;
end for;
end for;

3.5.2.2 EDGE FORMATION OF COMPATIBILITY GRAPH

The following pseudo code is for determining the edge matrix of the compatibility graph. There exist two cases to determine the incompatibility between the mappings or nodes of the compatibility graph.

Repeat for each node $i = 1$ to $n$ of compatibility graph
Repeat for each node $j = 1$ to $n$ of compatibility graph
If ($i \neq j$) then
If the node $i$ and $j$ are compatible then

$Compatibility_matrix[i*n+j] = TRUE;$

$Compatibility_matrix[j*n+i] = TRUE;$
endif;

endif;

end for;

end for;

Figure 3.13 The Contradictory edge mappings that should not have edge between them in compatibility graph.

The two cases, mentioned in figure 3.13 and 3.14 are the edges of compatibility graph that are to be discarded. In figure 3.13, the mappings between nodes 33-45 and 35-45 should not have edge in between them in the compatibility graph. Similarly the mappings between nodes 33-46 and 35-46 should not have edge in between them in the compatibility graph. Either the mappings 33-46 and 35-45 or the mappings 33-45 and 35-46 should be selected as part of the maximum clique. Hence the edge between the mappings 33-46 and 35-46 is discarded and the edge between the mappings 33-45 and 35-45 is discarded.
In figure 3.14, the cycle formation after merging is illustrated. If the edges between the mappings of the compatibility graph are supposed to have resulted with a cycle, then that edges between those mappings must be avoided.

Figure 3.14 Mappings that form a cycle and discarded edge in the compatibility graph.

The edge between the mappings 24-32 and 23-33 should be deleted as shown in figure 3.14b. Because if that edge is present, then both the mappings may be considered for the maximum clique and both of them are considered to be the merged nodes. If both
of them are considered for merging, then the cycle is formed between the nodes 24, 32 and 23, 33. Because there exist an edge in graph G3 from 23 to 24 and there exist an edge in graph G4 from 32 to 33. This cycle should be avoided. It can be determined by tracing the edges of the source graphs.

The graph G3 is traced down (traced up) by considering mapping 23-33, for whether the 24 is down to it and the graph G4 is traced up (traced down) for whether the 32 is up to it. If both are TRUE then, the edge between those mappings in the compatibility graph should be discarded in order to avoid the cycle in the resulting merged graph.

3.5.2.3 DETERMINATION OF MAXIMUM CLIQUE

The following is the pseudo code for determining the maximum clique in the Compatibility_matrix.

Repeat for each row \(i = 1\) to \(n\) of the Compatibility_matrix \((n \times n)\)

\[Clique_i = \text{NULL};\]

end for;

Repeat for each row \(i = 1\) to \(n\) of the Compatibility_matrix \((n \times n)\)

Insert \(i\) th element into the \(Clique_i\)

Repeat for all TRUE entries among \(n\) element in \(i\) th row

Find the element having the maximum no. of neighbors;

Check whether it is the neighbor of all the elements already inserted into the \(Clique_i\);

If so, insert that element into the \(Clique_i\);
Find the *maximum_clique* from the set of cliques determined above;

The clique can be determined by the information in the edge matrix of the compatibility graph. For each element the number of its neighbors is calculated. Then for each row the clique is determined. First the row element is added into the clique. Then all the elements in that row which should be TRUE are considered. Among those elements, the element which has maximum number of neighbors is first added, in to the clique. Then next element is chosen in the same fashion provided that should be the neighbor of all the elements already inserted into the clique. This should be repeated for all the elements in that row that are TRUE.

The above procedure is repeated for each row of the compatibility edge matrix. From the resultant set of cliques the maximum clique should be determined. The maximum clique is the clique that contains the maximum number of elements in it. After the identification of the maximum clique, the mappings in that clique are the merged nodes and edges of the merged graph. The nodes which are not merged are later added into the merged graph. The multiplexers are added appropriately to facilitate the data flow of the merged graph.

**3.5.2.4 MULTIPLEXER**

The multiplexers are appropriately added after the merging of two graphs in order to facilitate the data flow correctly. If two graphs are merged, the requirement of the multiplexer is checked. The multiplexer’s requirement is determined by analyzing the
inputs of the merged nodes of the two graphs. There arise many cases if the inputs of the merged nodes are considered. Major cases are whether two multiplexers are required or one or none.

Two multiplexers are required if the inputs are from different sources in graph $G_1$ and graph $G_2$ and they are not merged in the merged graph. It is the (i) case, shown in the figure 3.15. Then the cases (ii) and (iii) are, if one of the inputs is from the out side of the graphs and remaining three are from the graphs $G_1$ and $G_2$. Similarly if both the inputs from one graph are from outside and both the inputs from another graph are from different sources, then two multiplexers are needed. It is shown in the cases (iv) and (v).

Figure 3.15 Cases where two multiplexers are needed.
Figure 3.16 Cases where single multiplexer is needed.

One multiplexer is needed if any of the cases occurs as illustrated in figure 3.16. The different cases are: two are from different sources and the other two are merged one, three are from outside and two of the inputs of one from each graph from outside and other two are not merged.

Figure 3.17 Cases where no multiplexer is needed.
Multiplexers are not required if the cases match the figure 3.17. If either the inputs from the two graphs are merged among themselves or all inputs are from outside or two of them are merged, one from each graph and other two are from outside, then no multiplexers are needed.

When the merging between an ordinary graph and a merged graph takes place, then already existing multiplexers of the merged graph should be analyzed before going for a new multiplexer. In that also different cases are possible. The two main cases are, the currently merging node may have either one multiplexer or two multiplexers already.

If one multiplexer already exists, then the inputs of the multiplexer are to be analyzed to determine the suitability of the existing inputs of multiplexer in merging with one of the current inputs. Otherwise, requirement of another multiplexer is determined depending upon the inputs.

If two multiplexers already exist in the merged graph, there is no need of adding another multiplexer, but it is to be checked to decide the category of the current inputs of node going to be merged as illustrated in figure 3.18 and 3.19.

Figure 3.18 Cases to be considered while merging the already merged graph(s).
Figure 3.19 Cases under which two multiplexers already exist (after merging).

The figure 3.18 shows the situation of the node to be merged in the already merged graph. It may contain either one multiplexer or two or none before merging. The currently merging node is the one denoted by the circle. If two multiplexers already exist, it will fall under any one of the cases shown in the figure 3.19. In these cases, it is assumed that the graph G1 is the merged graph and G2 is a non-merged graph.

Case 1.1 is the one in which two multiplexers already exist in merged graph G1. The two inputs to the merging node from non-merged graph G2 are merging with the multiplexer inputs of the merging nodes from already merged graph G1. In this case, the inputs may be from the template itself or from outside the template. If the inputs are from outside the template then it is represented by mentioning it as 0.

In case 1.1.1 one of the inputs in both multiplexers just before the merging node of already merged graph G1 is 0. Both the inputs to the merging node from the non-
merged graph G2 are also 0. Hence those inputs merge with one another. Similarly, in case 1.1.2, one of the inputs in one multiplexer just before the merging node is 0 and one of the inputs of the merging node from the non-merged graph G2 is also 0. Hence these inputs get merged. One more input from both the graphs is not 0, but they are merged with each other.

Both the inputs from the non-merged graph G2 to the merging node are within the template and hence they are not 0 in case 1.1.3. One input in each of the multiplexer just before the merging node from merged graph G1 are merging with those inputs from G2. In case 1.2, the inputs from the non-merged graph G2 to the merging node is not 0. Also, they are not merging with any of the inputs of the two multiplexers of the merging node from the merged graph G1. Hence these inputs from G2 are added as one more input to the multiplexers of the merging node of G1.

Case 1.3 is the combination of case 1.1 and case 1.2. One of the inputs of the merging node of G2 is merging with one of the inputs of any one multiplexer just before the merging node of the merged graph G1. The other input is not merging and hence it is added as one more input of the other multiplexer just before the merging node of G1.

If one multiplexer already exists then, any one of the cases may match as shown in the figure 3.20. In case 2.1, one of the inputs of the merging node from the graph G2 is merging with the one input which is not having the multiplexer of merging node from G1. The other input is merging with one of the inputs of the only one multiplexer just before the merging node of G1.
In case 2.2, one of the inputs of the merging node from the graph G2 is merging with the one input which is not having the multiplexer of merging node from G1. The other input is not merged with any of the inputs of the only one multiplexer just before the merging node of G1. Hence that input is added as one more input in the only multiplexer of the merging node of merging node G1.

In case 2.3, one of the inputs to the merging node from the graph G2 is merging with the one of the inputs of the only multiplexer just before the merging node of G1. The other input is not merging and hence the second multiplexer is inserted between the merging node and that input. Both the non-merging inputs to the merging node from G1 and G2 are added as the inputs to the newly added multiplexer and the merging node now is having two multiplexer just before it.
In case 2.4, both the inputs are not merging and only one multiplexer already exists. Then one of the inputs to the merging node from the graph G2 is added as one more input to the already existing multiplexer. Then one new multiplexer is inserted between the merging node and its input does not have the multiplexer. The other input to the merging node from G2 is added to the input newly added to the multiplexer.

If there is no multiplexer before the merging node of the already merged graph, then it is considered as a new merging and it may match with any of the categories of figures 3.15 – 3.17.

3.5.2.5 THE MERGED GRAPH

The information about each merged node can be collected from the compatibility graph and the two source graph which are subjected to merge. Using that information, the inputs of the merging node from both the source graphs are identified. With that, the need of multiplexers is determined. For each merging node utmost 2 multiplexers are permissible, since only 2 inputs are allowed for an operation (node).
Figure 3.21. The structure of the merged graph.
There is no restriction in the number of inputs to the multiplexers. If the inputs to the merging node are merging themselves then there is no need of multiplexers. If the merging is between an already-merged graph and a non-merged graph, then analyzing of already existing multiplexers is necessary. The structure of the merged graph after the inclusion of multiplexers appropriately is shown in figure 3.21. The resulting merged graph is having the pointers to its list of nodes, edges and multiplexers. Each node and edge are having pointers to the list of merged nodes and edges merging with themselves respectively.

*A complete example of datapath merging:*

![Diagram of two graphs](image)

Figure 3.22 Graph1 and Graph2 that are subjected to data-path merging.

The figure 3.22 shows, the given two graphs G1 and G2 are analyzed to find out the similarity between them. The common nodes can be identified by their operation. The common edges are identified in the same way. If the source and destination nodes are
common or merged with each other, then the edges are merging. All the possible merging
nodes for each node of graph1 in graph2 and all the possible merging edges for each edge
of graph1 (G1) in graph2 (G2) are found.

Figure 3.23 All possible mappings of vertices and edges of Graph1 and Graph2.
Figure 3.24 The compatibility graph and its corresponding maximum clique.

The common nodes and edges are mapped as shown in figure 3.23 and its corresponding compatibility graph is shown in figure 3.24. For example, node 33 is multiplication operation of G1 which can be merged with node 45 or 46 of G2. So the mapping is between 33 and 45 and another one is between 33 and 46. Similarly the edge 33 – 34 of G1 is merging with 45 – 47 and the edge 35 – 36 of G1 is also merging with 45 – 47 of G2.

The representation of mapping is as follows:

33-*-1-1 is actually the node number, operation type, to which graph it belongs to and finally the occurrence of the operation in that graph. For example, 33 is of type
multiplication occurs first in graph G1 and node 35 is of same type multiplication occurs second in graph G1. Hence in node 35 the last field is 2. This numbering is needed to build the compatibility graph.

Figure 3.25 Merged graph of Graph1 and Graph2 and same with multiplexers.

The compatibility graph is constructed by considering all the node and edge mappings as the nodes of compatibility graph. Initially all the nodes are completely connected. The edges are possible between the nodes of the compatibility graph only if they are compatible with each other. That is, the mapping in one node does not contradict with the other. For example, 33 is mapping with 45 in one node of compatibility graph and again 35 is mapping with same 45 in another node. This is contradictory and the edge
between those nodes is not possible. Similarly all the edges between the nodes of compatibility graphs are checked. If the compatibility property is violating then the edge is discarded.

After discarding all the incompatible edges, the maximum clique is identified. Maximum clique is a sub graph of compatibility graph, in which the nodes are completely connected like a mesh and also have maximum number of nodes of the compatibility graph. After the maximum clique is obtained, the nodes of the maximum clique form the part of the merged graph. The entire merged graph is constructed by including the remaining nodes and edges that are not in the maximum clique but in the graphs G1 and G2. The case (iii) of 1 multiplexer needed is occurring before node 36,47 in figure 3.25. The case (v) of 1 multiplexer needed is occurring before node 37,48 in figure 3.25.

* A special case in datapath merging: *

![Graph 3](image1)

![Graph 4](image2)

Figure 3.26 Graphs to demonstrate special case of merging.
Figure 3.27 Special case in compatibility graph.

The graph3 and graph4 that are shown in figure 3.26 are the graphs used to demonstrate the special case of compatibility. In figure 3.27 the special case of compatibility between the edges of the compatibility graph is illustrated. The edge between the mappings 24-32 and 25-32 should be deleted, because of the contradiction between those mappings in the compatibility graph shown in figure 3.27b. But the edge between the mappings 24-32 and 23-33 is also deleted. This is because, if those nodes are merged, then that will lead to a cycle from 23 to 24 of Graph3 and 24(32) to 23(33) of
Graph 4. The cycle should be avoided in order to prevent the infinite execution. Hence this cycle is also considered while constructing the edges of the compatibility graph. This case is not discussed in earlier works[47].

This cycle can be determined by tracing the two graphs which are subjected for merging. Each vertex mapping of the compatibility graph should be checked with every other vertex mapping by tracing the path in which they occur in their graphs.

### 3.6 SCHEDULING OF BEST CCHBS

The identified profitable CCHBs are ordered according to their execution dependency and merged together. Merged CCHBs from different applications are arranged based on their gain factor (GF). Gain Factor is decided based on the performance factor needs to be improved. In this work the gain factor is calculated as given below.

![Figure 3.28 Coordination and scheduling of CCHBs](image-url)
Gain Factor = CCHB_FPGA size ratio + Speedup + \((\text{software cycles saved} \times \text{no. of occurrences})/\text{Total application software cycles}\) + no. of operations merged.

The above is to increase the device utilization, i.e., reduces the occupancy of FPGA by the CCHBs of the application. Gain Factor for improving the processing time is different in which parameters like CCHB-FPGA size ratio and no. of operations merged are of less weight.

The first set of merged CCHBs of different applications is determined and configured into the FPGA. If the FPGA could not accommodate all the merged CCHBs, then wait for the completion of execution as shown in figure 3.28. Since runtime reconfiguration is simulated, when even a part of the circuit execution is over, the FPGA can be reconfigured partially to accommodate the remaining CCHBs while the rest is still running. On the other hand, after the first set of CCHBs is scheduled successfully, the entire process is repeated till the end of all the applications execution.

The scheduling is done with the gain factor considered and the execution order of the applications. As applications are profiled through Trimaran compiler, scheduling is performed at compile time. The scheduler checks for the same CCHBs across different applications in exhaustive way to further reduce the device utilization.

3.7 HDL CODE FOR SCHEDULED CCHBS

Verilog hardware definition language code for each template is stored in template library storage. For each scheduled CCHB, its corresponding generalized code is made available. Those HDL codes are tailored for the given application, synthesized properly and configured into the FPGA. The software implementation of the application system
should be correspondingly modified in order to execute the computationally intensive parts in FPGA.

### 3.8 MIBENCH EXAMPLE APPLICATION

Templates and matches of Mibench benchmark application: AdpcmDec

<table>
<thead>
<tr>
<th>No.of Matches</th>
<th>10</th>
</tr>
</thead>
</table>

**Template no. 1**

- Application name: AdpcmDec
- Function name: _adpcm_decoder
- Basic block: 6
- Template_id: 1
- No.of matches: 1
- No.of Nodes: 2

<table>
<thead>
<tr>
<th>match_id</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match_id within template:</td>
<td>1</td>
</tr>
<tr>
<td>DFG nodes:</td>
<td></td>
</tr>
<tr>
<td>node:0 level:0 type:SHL_W</td>
<td></td>
</tr>
<tr>
<td>node:1 level:1 type:ADD_W</td>
<td></td>
</tr>
</tbody>
</table>

**Template no. 2**

- Application name: AdpcmDec
- Function name: _adpcm_decoder
- Basic block: 24
- Template_id: 1
- No.of matches: 1
- No.of Nodes: 2

<table>
<thead>
<tr>
<th>match_id</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match_id within template:</td>
<td>1</td>
</tr>
<tr>
<td>DFG nodes:</td>
<td></td>
</tr>
<tr>
<td>node:0 level:0 type:SHL_W</td>
<td></td>
</tr>
<tr>
<td>node:3 level:2 type:ADD_W</td>
<td></td>
</tr>
</tbody>
</table>

**Template no. 3**

- Application name: AdpcmDec
- Function name: _adpcm_decoder
Basic block : 10
Template_id : 2
No.of matches : 1
No.of Nodes : 2

match_id : 3
Match_id within template : 1
DFG nodes :
node:1 level:1 type:AND_W
node:2 level:2 type:AND_W

---------------------------------
Template no. : 4
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 5
Template_id : 3
No.of matches : 1
No.of Nodes : 2

match_id : 4
Match_id within template : 1
DFG nodes :
node:4 level:3 type:SHRA_W
node:5 level:4 type:AND_W

---------------------------------
Template no. : 5
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 5
Template_id : 4
No.of matches : 1
No.of Nodes : 2

match_id : 5
Match_id within template : 1
DFG nodes :
node:3 level:2 type:EXTS_B
node:4 level:3 type:SHRA_W

---------------------------------
Template no. : 6
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 5
Template_id : 48
No.of matches : 1
No.of Nodes : 3

match_id : 6
Match_id within template : 1
DFG nodes :
node:3 level:2 type:EXTS_B
node:4 level:3 type:SHRA_W
node:5 level:4 type:AND_W

Template no. : 7
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 15
Template_id : 5
No.of matches : 1
No.of Nodes : 2

match_id : 7
Match_id within template : 1
DFG nodes :
node:0 level:0 type:SHRA_W
node:2 level:2 type:ADD_W

Template no. : 8
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 13
Template_id : 5
No.of matches : 1
No.of Nodes : 2

match_id : 8
Match_id within template : 1
DFG nodes :
node:0 level:0 type:SHRA_W
node:2 level:2 type:ADD_W

Template no. : 9
Application name : AdpcmDec
Function name : _adpcm_decoder
Basic block : 2
The large circles in figure 3.29 are templates and small circles are matches. The identified profitable matches are 1, 2, 9, 7, 8, 4, 3. The above sets of matches identified are
from the match set which consists of matches subsumed in other matches. Eg. Matches 4 and 5 are subsumed in match 6.

The conflict graph after removing subset matches is shown in figure 3.30. The process is performed by removing subsumed matches. The identified profitable matches are 1,2,9,7,8,6,3. The order of execution of these CCHBs is 9,6,1,3,7,8,2.

The seven profitable CCHBs identified with conflict graph approach are subjected to merging as shown in figure 3.31a. CCHB 9 is the first one and it is not merged with the next CCHB 6. Similarly the successive CCHBs are not having any common operations. Only CCHB 7 and CCHB 8 are merging together as shown in figure 3.31b.
Merged CCHBs from different applications are scheduled onto the FPGA based on the gain factor. When multiple CCHBs are competing, CCHB with highest gain factor is added into configuration. If enough logic cells are free, next highest CCHB is scheduled. If same CCHBs are arriving from different applications then their gain factors are added and scheduled.

![Diagram](image)

Figure 3.31 a) profitable CCHBs and b) merged CCHBs of example application

CCHBs are taken from four benchmark applications such as Adpcm Dec., Adpcm Enc., Blowfish and Sha from Mibench benchmark suite to analyze this framework. It occupies certain number of logic cells in the FPGA to perform its operation. More than one CCHB can be configured based on the logic cell availability in the FPGA. If the FPGA cannot accommodate all the CCHBs, then after the execution completion by
currently configured CCHBs, the waiting CCHBs is reconfigured. From the study of four applications from benchmark suite, it is observed that the device utilization is increased with size of CCHBs not greater than 0.5 times or not less than 0.85 times of the size of FPGA.

3.9 CONCLUSION

The complete framework of coordinating CCHBs for FPGA accelerated system is explained in this chapter. Computation intensive part of the application system is configured as CCHBs in FPGA to accelerate it. Hence the process of scheduling is necessary to improve the performance. The mechanism for coordinating and managing the CCHBs is implemented using the two heuristics. The maximum independent set problem is resolved with the conflict graph approach and the maximum clique problem is solved using compatibility graph approach. The next step is to test and analyze the results of the framework by applying them on real time application systems which is explained in chapter 4.