Generalized Resource Sharing

S. Raje R. A. Bergamaschi

IBM T. J. Watson Research Center, NY, USA

Abstract

Resource sharing is one of the main tasks in high-level synthesis, and although many algorithms have addressed the problem there are still several limitations which restrict the generality and applicability of current algorithms. Most clique-partitioning-based algorithms use local and inaccurate cost-functions which result in inefficient results. This paper presents algorithms for the resource sharing problem on registers and functional units, and shows how they overcome the limitations of existing algorithms.

The main characteristics of this work are: interleaved register and functional unit merging in a global clique partitioning based framework, accurate merging cost estimation, accurate interconnect cost estimation, relative control cost taken into account, and efficient false loop elimination. The results obtained show significant improvements in the delay of designs, while also minimizing area, specially for large designs with many sharing possibilities.

1 Introduction

Resource sharing is one of the main problems in high-level and register-transfer-level synthesis. The problem consists in minimizing the number of registers and functional units used in a design with the primary goal of minimizing area. Extra cost metrics such as delay and power have also been considered.

Several algorithms have been proposed for sharing registers and functional units. These algorithms can be broadly classified into *constructive* or *global* algorithms. Constructive algorithms, such as EMUCS [1], build the data-path one element at a time. Global algorithms [2], are usually based on *clique partitioning* techniques. The registers or functional units to be merged are mapped onto a *Compatibility Graph*, which is partitioned into cliques. The advantages of this approach are that it allows for more global solutions than the (greedy) constructive approaches, and tend to produce better results. Although clique partitioning techniques for resource sharing have been widely adopted by several systems, strong limitations which have not yet been addressed still remain. This paper presents a generalized formulation and algorithms for global clique-partitioning-based resource sharing which overcome many of the limitations of current algorithms. It also detects and eliminates false loops, possibly introduced by sharing, more efficiently than previously published. The following paragraphs explain these limitations and introduce the algorithms in this paper which overcome them.

Sharing order: In all algorithms to date, sharing of registers is done separately from sharing of functional units, that is, either register sharing is done first followed by functional unit sharing (as in Facet [2]), or vice-versa. This is inefficient because it directly impacts the accuracy of the cost computation. The algorithm presented in this paper shares both registers and functional units in the same clique partitioning formulation.

Cost computation: In most systems, the cost associated with merging resources is a purely local metric, which can severely impact the advantage of a global clique partitioning approach, and produce poor results.

In this paper, the primary cost metric is based on a projected area savings that would result if the whole clique were actually formed. That is, instead of measuring the cost of a single merging, the algorithm computes the final cost as if the whole clique is merged. Moreover, this cost takes into account actual technology-dependent values and accurate interconnection costs (using multiplexer-based point-to-point interconnections).

Interconnect costs: When sharing two elements, one can compute the resulting interconnection costs based on the inputs and outputs of the two elements at that stage. However, future mergings involving elements connected to the inputs or outputs of the merged nodes may change the interconnection cost, invalidating the original computation.

In this paper, the cost function also considers whether the inputs and outputs of the two elements being merged are compatible. If so, they may be merged in the future, which will reduce the interconnection cost of the current merging. This *compatibility look-ahead* mechanism is called *Future_Connectivity.*

Another source of inaccuracy is the handling of commutative inputs to functional units. The approach presented in this paper allows all elements to swap all inputs every time a new element is added to a clique.

False loop elimination: The algorithm in [3] identifies a number of potential loop-forming mergings and removes the corresponding compatibility edges from the compatibility graph. This algorithm can be very inefficient because it removes far more compatibility edges than really needed, which in turn prevents possible cost-effective mergings.

The algorithm in [4] uses zero-one quadratic programing to do resource sharing. It eliminates false

0-89791-993-9/97 $10.00 © 1997 IEEE

loops by imposing additional constraints to their mathematical program. Additional constraints costs run-time, which is already high for algorithms that use quadratic programing.

The false loop elimination algorithm described in this paper is more efficient because it is embedded within the clique partitioning algorithm. Once a compatibility edge is selected for merging, the algorithm checks whether that merging will definitely cause a false loop by doing a topological analysis of the resulting network, and if so, the merging is not carried out and the edge is deleted. In this way, only a minimal number of edges is removed from the compatibility graph, thus allowing for more mergings to be considered.

Control logic effects: Resource sharing is usually done with little or no knowledge of the effects on control area. In general, control area is difficult to estimate because of the logic optimizations of the controller during the logic synthesis step.

The resource sharing algorithm presented in this paper looks at the control equations associated with each data-path element and uses that as a secondary decision metric when selecting elements to be merged.

2 Framework

The starting point for this resource sharing algorithm is an initial data-path and controller generated from a scheduled control and data flow graph [5]. Based on this initial data-path and controller, and the scheduled control/data flow graph, the complete *compatibility graph* (CG) is derived. This graph is the union of the compatibility graphs for registers and functional units.

Figure 1 shows a simple VHDL description and the corresponding scheduled control/data flow graph. This example uses "*wait until not clock'stable and clock='1'*" statements to represent either the scheduled states (as given by a scheduling algorithm) or states explicitly declared by the designer, resulting in a finite-state machine controller with five states. The initial data-path and controller and compatibility graph for this example are given in Figure 2, where *o1* to *o5* are operations and *a*, *b*, *c*, *d* are registers. Among the control signals generated by the controller there are some which are needed in the initial data-path (e.g., *ld_a*, represented by solid arrows) and some which are *potential* control signals (represented by dashed arrows), that are not needed in the initial data-path but may be needed if elements are shared. For example, signal *o1_sel* associated with adder *o1* is not yet needed because *o1* is a single function operator. However, if *o1* is merged with *o5* then both controls *o1_sel* and *o5_sel* will be needed to select the different functions in the resulting ALU. This resource sharing algorithm looks at both types of controls for determining control costs as well as detecting false loops due to sharing.

3 Cost Computation

The heuristic for forming cliques is fully based on costs. These costs are not necessarily proportional to

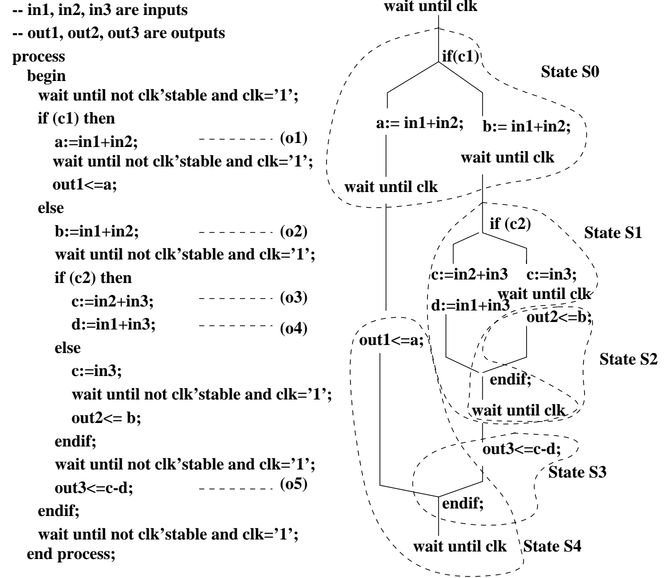

Figure 1: Behavioral Description and corresponding scheduled Control/Data-flow graph

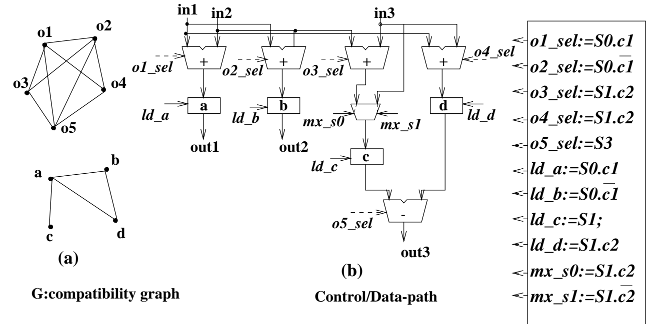

Figure 2: (a) Compatibility graph (b) Initial data-path and controller.

the size of the cliques or to the minimum number of cliques. The partitioning algorithm selects compatibility edges for merging based on the costs of all edges at any time. As a result, several cliques may be formed concurrently, instead of the traditional clique heuristic which forms cliques one-by-one with the goal of minimizing the cost of the clique being formed and the total number of cliques. The goal of this algorithm is to minimize the cost of all cliques together, which may or may not result in the minimum number of cliques. The primary cost is measured as area, but by accurately estimating interconnection costs, the interconnections are minimized, which has a direct effect on delay.

This algorithm uses a single compatibility graph which is a union of the functional unit compatibility graph and the register compatibility graph. Edges connecting registers or functional units may be selected at any time and merged in different cliques, effectively interleaving register and functional unit

merging. This allows for more accurate estimation of the interconnection costs and makes the algorithm less dependent on the order in which registers and functional units are shared.

The partitioning algorithm works iteratively by selecting a compatibility edge based on the cost function and merging the nodes connected by the edge.

The cost function is actually not a single number but a decision tree based on different metrics. The decision tree for selecting an edge works as follows:

Step 1: Compute the *Projected_Area_Savings* of all compatibility edges. Select the subset of edges whose *Projected_Area_Savings* fall within $N\%$ of the top savings value. The value for N was empirically chosen to be around 80%. These initial set of edges then go to the secondary selection criteria.

Step 2: Compute the *Future_Connectivity* of all edges selected in Step 1. Select the subset of edges with *Future_Connectivity* within 80% (empirically chosen) of the best value. Pass this subset of edges to the tertiary selection criteria.

Step 3. If more than one edge is left in Step 2, compute the *Control_Similarity* of those edges and select the edge that has the highest *Control_Similarity*. Remaining ties are broken by using the highest *Projected_Area_Savings* followed by the highest *Future_Connectivity*.

3.1 Projected Area Savings

The *Projected_Area_Savings* for a compatibility edge estimates the area savings resulting from merging, not only the two nodes (connected by the edge) but also any other node that could end up in the same clique. In other words, it estimates the cost of the full clique, if formed. This helps overcome the locality effect of computing the cost based on the two end nodes alone.

This saving is the product of the total area savings (active area and interconnection area saved) when merging the two end-nodes of an edge and the number of common neighbours to the two nodes being merged, as described in the formula:

$$Projected_Area_Savings(e) = (Active_Area_Savings(e) + Connectivity_Savings(e)) * (Commonality(e) + 1)$$

Algorithms for computing Active_Area_Savings(e) and Connectivity_Savings(e) are given in the following sections. Commonality(e) is the number of common neighbours to both end nodes of edge e. Although there is no guarantee that the whole clique will be formed, this projected area is a more global cost than the edge cost alone.

3.1.1 Active Area Savings

Active_Area_Savings is defined as the difference between the sum of the areas for the two end-nodes and the area of the shared resource that would implement the operations in the two nodes if the edge were to merge. In the formula below, edge e connects nodes v_i and v_j:

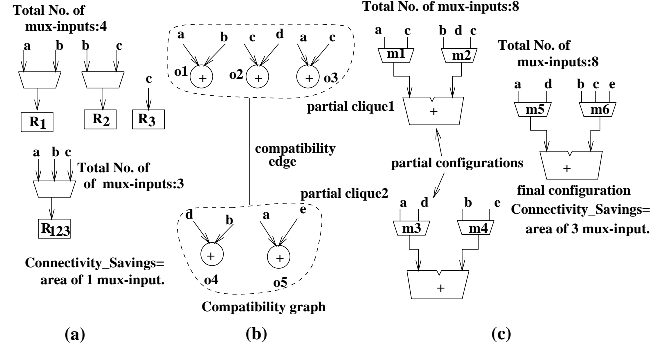

Figure 3: (a) Computing Connectivity_Savings for registers (b) Compatibility graph for operations and initial multiplexer configurations for the partial cliques. (c) Multiplexer configuration for the final clique and computation of Connectivity_Savings for operations.

$$Active_Area_Savings(e) = area(v_i) + area(v_j) - area(v_i, v_j)$$

The area values are computed based on a technology library using the estimation algorithms described in [6].

3.1.2 Connectivity Savings

When two nodes are shared, the inputs have to be reconnected to the merged node possibly causing the creation of multiplexers. Connectivity_Savings is defined as the difference between the sum of the interconnections to the two elements and the final interconnections to the merged element, where interconnections are implemented as point to point multiplexers. The unit used is not number of inputs but the actual area of the multiplexers needed.

$$Connectivity_Savings(e) = mux_area(v_i) + mux_area(v_j) - mux_area(v_i, v_j)$$

Calculation of Connectivity_Savings for edges with register end nodes is straightforward and is derived from the current data path. Special care is taken to consider the actual inputs independently of any existing multiplexer. In Figure 3a for example, the algorithm looks at the multiplexer inputs connected to R_1 and R_2 and at the final multiplexer connected to the merged node R_{123}. Trees of muxes in the initial data-path are collapsed into single-level muxes, so that the input count reaching the registers or operations is accurate.

Calculation of Connectivity_Savings for operations is a more involved procedure. It is critical to the accuracy of this computation to be able to permute inputs to commutative operations, as this affects directly the required number of inputs to the multiplexers.

Most algorithms use a purely greedy permutation each time a new element is added to a clique. Suppose that operations *o1* to *o5* in Figure 4 are gradually combined into a clique one by one. Using a greedy permutation of the new element inputs leads to suboptimal results as indicated in the top part of Figure 4.

The algorithm used in this paper permutes all the inputs each time a new element is added to the clique

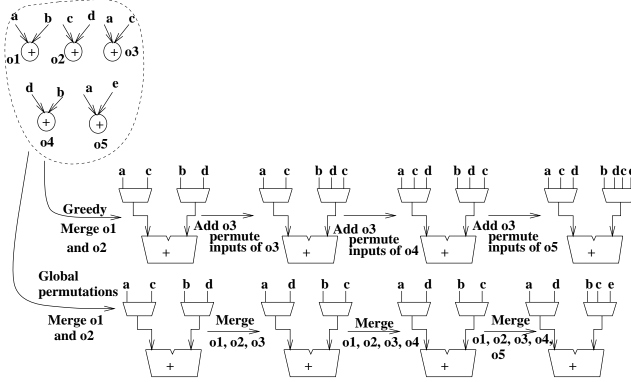

Figure 4: Comparing greedy and global input permutations on commutative operations.

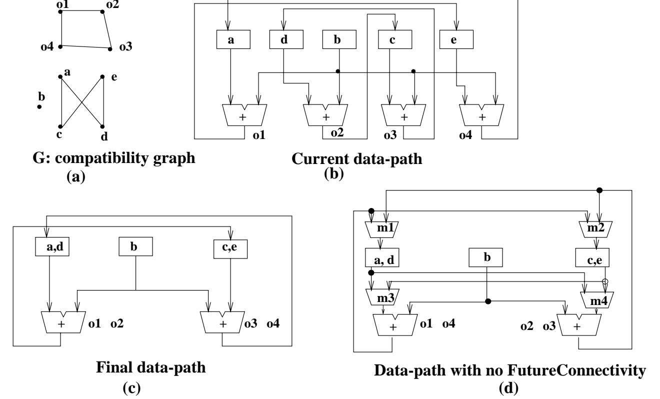

Figure 5: (a) Compatibility graph (b) Initial data-path (c) Final data-path using Future_Connectivity as a criteria (d) Data-path using only Connectivity as the criteria

and is able to find the optimal permutation resulting in the minimum number of multiplexer inputs, as indicated in the bottom part of Figure 4. For a full detailed description of the algorithm , see [7].

As an example of Connectivity_Savings for operations, consider the compatibility graph in Figure 3b, in which two partial cliques have already been formed and the algorithm needs to compute interconnection costs involved in merging them. The Connectivity_Savings is the difference in the areas of input multiplexers to both partial cliques and the areas of the input multiplexers in the final merged clique. Permuting the inputs of all operations using the algorithm above results in the multiplexers shown in Figure 3b (for the partial cliques) and Figure 3c (for the merged clique). The Connectivity_Savings equals the area of three fewer multiplexer inputs.

3.2 Future Connectivity

Future_Connectivity is a measure of savings in interconnections due to future mergings involving inputs and outputs of the current nodes being merged. Distinct inputs and outputs of two nodes may eventually be merged into a single resource; this is not reflected in the Connectivity_Savings number for an edge but is captured in the Future_Connectivity.

Figure 5 shows an initial data-path and the associated compatibility graph, where a, b, c, d, e are registers and $o1$, $o2$, $o3$, $o4$ are operations, all of which have a common input b.

The Connectivity_Savings for each of the compatibility edges $(o1, o2)$, $(o1, o4)$, $(o2, o3)$, $(o3, o4)$ is -2 multiplexer inputs, as a 2-input multiplexer will be required for any of these mergings (and there were no multiplexers in the initial data-path). Connectivity_Saving therefore cannot decide on which edge is better for merging (note that the Projected_Area_Savings are also the same for all edges).

Future_Connectivity looks at the registers connected to the inputs of the operations and checks if they are compatible. Register a and d are compatible and could eventually be merged, whereas register a and e are not compatible and could not share the same register. It follows that preference should be given to merging operations $o1$ and $o2$ instead of operations $o1$ and $o4$ because the inputs to $o1$ and $o2$ are likely to be merged in the future resulting in fewer interconnections (when compared to merging $o1$ and $o4$). Similar arguments can be made about merging $o1$ and $o2$ over $o2$ and $o3$. Future_Connectivity is a measure of the above preference.

Future_Connectivity for an edge is measured as the number of compatible edges among the nodes that form the left input set plus the number of compatible edges among nodes that form the right input set plus the number of compatible edges among outputs of the nodes being merged. For operations, this measure takes place after a "good" permutation of inputs is found, (see [7]).

In Figure 5, edge $(o1, o2)$ has Future_Connectivity of 1 (since a is compatible with d), whereas edge $(o1, o4)$ has Future_Connectivity of 0. Using Future_Connectivity as a criteria to merge edges the algorithm results in a final data-path as shown in Figure 5(c). Figure 5(d) shows a possible outcome for the data-path when Future_Connectivity is not considered as one of the merging criteria.

3.3 Control Similarity

Control_Similarity measures the amount of commonality between the control conditions associated with the execution of the two end operation nodes of an edge or the load-enable conditions of the two end register nodes. This commonality is measured as the number of common support variables in the control conditions for the two nodes. The support variables are extracted from the BDDs representing the control conditions. These conditions can either be *active* conditions already present in the initial data-path or *potential* conditions activated only when nodes are merged.

Merging of functional units or registers may result in extra control logic to be created. In the case of registers, this extra control logic is represented by the load_enable of the merged register (which should be

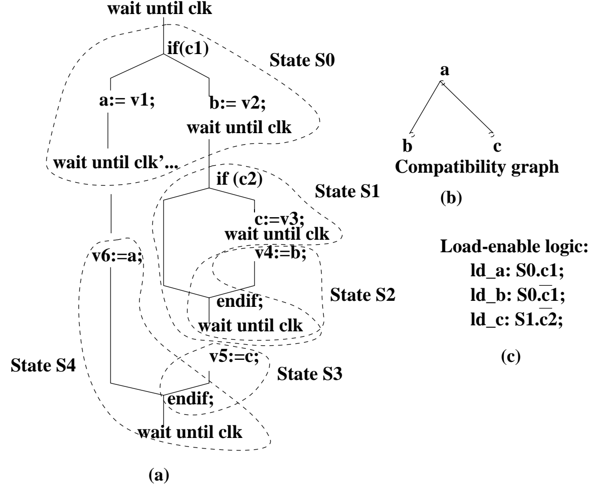

Figure 6: (a) Control/data-flow graph (b) Compatibility graph for the registers (c) Control Equations for the load-enable signals for registers

the OR of the original load-enable's) and by the select signals on input multiplexers. In the case of functional units, the extra control logic is represented by the select signals on the resulting ALU (if different functions are merged) and by the select signals on input multiplexers. The higher the number of common support variables in the *OR'ed* controls, the higher the probability that the resulting control can be optimized.

Figure 6a shows a fragment of a scheduled control/data flow graph. This example has 5 states, $S0$ through $S4$. The compatibility graph for registers a, b, and c is given in Figure 6b and the load-enable conditions, shown in Figure 6c, are a combination of the state value and the conditions on the branches of the control/data flow graph.

In this case the Projected_Area_Savings and Future_Connectivity are the same for both edges, (a, b) and (a, c). However, if a and b share a single register the resulting load-enable control of this register $R_{a,b}$ will be $S0.c1 + S0.\overline{c1} = S0$, whereas if a is shared with c the load-enable control of register $R_{a,c}$ will be $S0.c1 + S1.\overline{c2}$ which requires more gates to be implemented. The edge (a, b) should therefore be preferred over the edge (a, c) for merging. The Control_Similarity metric represents such preference.

In Figure 6, the Control_Similarity for edge (a, b) is 1 ($c1$ is a common support variable) and the Control_Similarity for edge (a, c) is 0. Note that since the state value is encoded in a state register which is common to all control signals, the state variable is not counted.

4 False Loop Elimination

The elimination of false loops is embedded within the sharing algorithm. No preprocessing is done to remove edges that could potentially lead to false loops as in [3]. When an edge is finally selected for merging (based on the cost criteria presented earlier), the

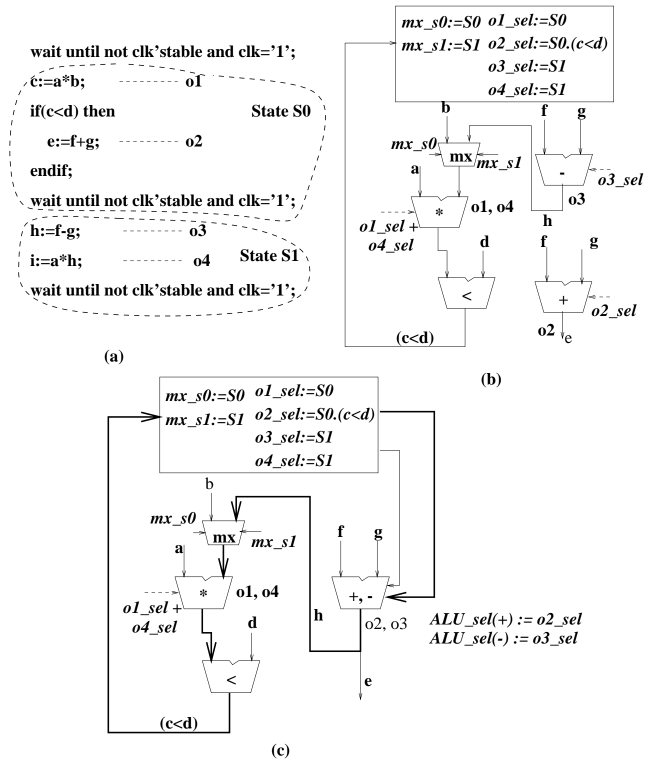

Figure 7: (a) Behavioral description, (b) Data-path and controller, (c) Data-path and controller if operations $o2$ and $o3$ are merged.

algorithm checks if merging the two nodes will cause a false loop in the design. If it does, then the edge is discarded and removed from the compatibility graph. The algorithm to detect false loops in the design essentially traverses the data-path graph and the controller (both *active* and *potential* controls). A *depth-first search* is performed starting from the output of one of the nodes, and follows the combinational data and control paths. If the traversal reaches the other node then a false loop exists, in which case the edge is discarded.

Figure 7 shows a fragment of a behavioral description and the corresponding data-path and controller, and suppose that $o1$ and $o4$ have already been merged. Operations $o2$ and $o3$ have the same inputs and can also be shared. Based on the Active_Area_Savings and Connectivity_Savings these two operations are good candidates for merging. The false loop detection algorithm starts the depth-first traversal on the output of $o3$, which leads to multiplexer mx, to multiplier $o1, o4$, to the comparator and into the controller. As the output of the comparator is in the support of control signal $o2_sel$, the traversal follows $o2_sel$ reaching adder $o2$. Operation $o2$ is therefore reachable from operation $o3$ and merging the two would lead to a false loop in the design.

The above false loop elimination approach removes far fewer compatible edges than the one described in [3]. An edge is removed only once it is selected for

merging and it is known for certain that merging the nodes leads to a false loop in the design. This technique allows for more cost effective mergings.

5 Generalized Resource Sharing: Complete Algorithm

Each step in the algorithm has already been explained in the previous sections. The pseudo-code below gives the overall sequence of steps involved in the complete algorithm.

```
Generalized_Resource_Sharing
(Compatibility graph CG,
 Initial data-path DPG,
 Initial Controller CTR) {
  while (edge_in_CG) {
    e = Select_Edge_to_Merge(CG, DPG, CTR);
    if (e == NULL) break;
    if (False_Loop_Detection(e, DPG, CTR))
      remove_edge (e, CG);
    else {
      merge_nodes(e);
      update_compatibility_graph(G);
      update_data_path_controller(DPG, CTR);
    }
  }
}
```

Procedure **Select_Edge_to_Merge** computes all the area, interconnections, future connectivity and control costs in all edges and selects one based on the selection criteria described in section 3. Then procedure **False_Loop_Detection** is called to check if merging the selected edge causes a false loop; in which case the edge is removed from CG. Otherwise the two end nodes of the edge are merged and the compatibility graph, the data-path graph and the controller are updated. This is repeated until no further edges are found in CG or no more edges can be found worthy of merging.

This algorithm does not require all edges in CG to be merged. If merging an edge is considered at some point in the algorithm to increase the cost permanently, then it is never selected. This is called a *stopping criteria*. If an edge has negative Projected_Area_Savings and has no common neighbours, it is highly unlikely that it will improve the cost of the design and thus it is not selected (Select_Edge_to_Merge returns null if all remaining edges in CG fall in this case).

6 Results

The generalized resource sharing algorithm has been implemented and tested with benchmarks and industrial examples. The results were compared against results produced by a coloring-based resource sharing algorithm [8] which aims primarily at producing minimum number of cliques, thus trying to minimize the number of functional units and registers. The algorithm in [8] does not attempt to eliminate false loops.

All examples were synthesized using both algorithms and the numbers of registers, functional units and multiplexers are compared. In addition, the results were submitted to logic synthesis for optimization and mapping and the final values of area and delay are compared.

For initial reference, Table 1 gives the total number of register bits, the number and type of functional units and the number of multiplexer bits present in the initial data path prior to resource sharing. Note that chaining is allowed, so operations can be connected to other operations or registers directly or through multiplexers.

Benchmarks	Initial Data Path		
	Reg bits	FUs	MUX bits
DIFFEQ	176	1(<), 2(+), 6(*), 2(-)	0
ELLIPTF	384	26(+), 4(*)	0
FRISC	96	12(+), 12(-), 3(=), 1(>)	1134
KALMANM	64	5(+), 10(-), 10(=), 1(<=), 5(*)	440
M6502	593	14(+), 6(-), 11(=), 2(>), 2(>=)	1484
MAHA	36	5(=), 8(+), 8(-)	64
RCV8251	19	2(-), 4(=)	53
Example 1	0	20(+), 24(=), 8(<), 8(>)	5192
Example 2	0	55(+), 22(-), 31(=), 2(<), 24(<=), 3(>), 24(>=)	3402
Example 3	0	16(-), 30(=), 4(/ =)	3104
Example 4	0	8(+), 4(=)	3998

Table 1: Initial data path: registers bits, functional units and input multiplexer bits. Each functional unit is a multi-bit operator.

All results were obtained by directing the two algorithms to merge only operations of the same kind, that is, adders with adders, subtracters with subtracters, etc. The algorithm can also handle merging of different operations in a multi-function unit.

Table 2 shows the number of register bits, the number and type of functional units, the number of multiplexer bits, and the area and delay values obtained after logic synthesis for the two resource sharing algorithms. Area and delay values are given in normalized area units (au) and delay units (du) respectively. Percentage improvements in area (%A) and delay (%D) are also given. From this table it can be seen that the algorithm in this paper produced better results in almost all cases, with several being significantly better in both area and delay.

The coloring algorithm [8] produced data-path with false loops in Elliptf, M6502 and Maha, so the delay values for these examples could not be computed reliably by static timing analysis. The generalized algorithm did marginally worse in Elliptf and Maha and better in M6502 despite removing some of the compatibility edges. The area results for Kalman show a significant increase because of an extra multiplier created by the generalized algorithm, which resulted from the elimination of an edge between two multiplication operations to avoid a false loop. The coloring algorithm did not create a false loop in Kalman due to a different merging order among edges which, by chance, did not produce a false loop.

Table 3 compares the number of edges removed by the generalized resource sharing algorithm to eliminate false loops against the number of edges removed by the algorithm in [3]. It can be seen that the gen-

Benchmarks	Coloring [8]					New Generalized Sharing Algorithm					% Improvements	
	Reg bits	FUs	MUX bits	Area	Delay	Reg bits	FUs	MUX bits	Area	Delay	%A	%D
Diffeq	80	1(<), 1(-) 1(+), 2(*)	336	4294	26.10	80	1(<), 1(-) 1(+), 2(*)	224	3942	24.33	8.2	6.8
Elliptf	160	2(+), 1(*)	720	5572	-	160	6(+), 1(*)	720	5998	25.02	-7.6	-
Frisc	96	2(+), 1(-) 3(=), 1(>)	990	5011	15.87	96	2(+), 1(-) 3(=), 1(>)	910	4786	15.36	4.5	3.2
Kalman	40	2(+), 2(-) 3(=), 2(*) 1(<=)	668	6677	31.47	24	3(+), 3(-) 5(=), 3(*) 1(<=)	496	8089	24.05	-21.1	23.6
M6502	119	4(+), 1(-) 4(=), 2(>) 1(>=)	1911	11373	-	110	5(+), 2(-) 5(=), 2(>) 1(>=)	1273	9484	17.28	16.6	-
Maha	12	5(=), 1(+) 1(-)	100	777	-	12	5(=), 2(+) 2(-)	112	801	11.83	-3.1	-
Rcv8251	11	1(-), 2(=)	76	1551	7.96	12	1(-), 3(=)	50	1366	6.17	11.9	22.5
Example 1	0	4(+), 24(=) 2(<), 2(>)	2292	17735	24.84	0	4(+), 24(=) 2(<), 2(>)	1940	16525	19.23	6.8	22.6
Example 2	0	15(+), 11(-) 1(<), 16(<=) 2(>), 16(>=) 31(=)	2682	50966	25.15	0	15(+), 11(-) 1(<), 16(<=) 2(>), 16(>=) 31(=)	2490	49099	20.29	3.7	19.3
Example 3	0	6(-), 29(=) 3(/ =)	1424	3862	15.91	0	6(-), 30(=) 3(/ =)	1360	3493	11.18	9.6	29.7
Example 4	0	4(+), 2(=)	1729	6252	9.45	0	4(+), 2(=)	1565	4686	5.17	25.0	45.3

Table 2: Resource sharing results for high-level synthesis benchmarks and industrial examples.

eralized resource sharing algorithm removes far fewer edges than the algorithm in [3].

The execution time for the generalized resource sharing algorithm ranged from 0.1 to 7.8 seconds for the example with the largest compatibility graph. The largest compatibility graph had over 2000 edges. Examples with several thousand edges have been synthesized in proportional run times. CPU times were measured on a IBM RS/6000 133 MHz workstation.

Benchmarks	Total No. of Compatibility Edges	No. of Edges Removed by [3]	No. of Edges Removed by the Generalized Sharing Alg.
Frisc	322	63	0
Kalman	376	178	4
M6502	526	157	2
Maha	111	42	5
Rcv8251	13	1	0

Table 3: Number of edges removed to avoid false loops.

7 Conclusions

This paper presented an efficient algorithm for resource sharing in high-level and RT-level synthesis. Algorithms were given for the most important problems in resource sharing, namely area and interconnection costs, control cost and false loop elimination.

Among the new concepts presented in this paper are: interleaved register and functional unit merging; accurate computation of interconnection cost, including global permutation of inputs; future connectivity costs based on the compatibility of inputs and outputs of the nodes being merged; control similarity costs; and efficient false loop elimination.

These algorithms were integrated into a generalized resource sharing system which produces results with comparable or better area and significantly better delay when compared to traditional resource sharing algorithms (targetted to minimizing the number of cliques).

References

[1] C. Hitchcock III and D. Thomas, "A method of automated data path synthesis," in *Proceedings of the 20th ACM/IEEE Design Automation Conference*, pp. 484–489, ACM/IEEE, June 1983.

[2] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, pp. 379–395, July 1986.

[3] L. Stok, "False loops through resource sharing," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 345–348, IEEE, November 1992.

[4] W. Geurts, F. Catthoor, and H. De Man, "Quadratic zero-one programming-based synthesis of application-specific data paths," *IEEE Transactions on Computer-Aided Design*, vol. CAD-14, pp. 1–11, January 1995.

[5] R. Bergamaschi, R. O'Connor, L. Stok, M. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao, "High-level synthesis in an industrial environment," *IBM Journal of Research and Development*, vol. 39, pp. 131–148, January/March 1995.

[6] R. Camposano and R. A. Bergamaschi, "Redesign using state splitting," in *Proceedings of The European Design Automation Conference*, (Glasgow, Scotland), pp. 157–161, IEEE, March 1990.

[7] R. A. Bergamaschi, *The Development of a High-Level Synthesis System for Concurrent VLSI Systems.* PhD thesis, University of Southampton, Southampton, England, February 1989.

[8] R. A. Bergamaschi, R. Camposano, and M. Payer, "Allocation algorithms based on path analysis," *INTEGRATION, the VLSI Journal*, vol. 13, pp. 283–299, September 1992.