Datapath Allocation

Zoltan Baruch

Computer Science Department, Technical University of Cluj-Napoca
e-mail: *baruch@utcluj.ro*

Abstract. The datapath allocation is one of the basic operations executed in the process of high-level synthesis. The other operations are partitioning and scheduling. The datapath allocation problem consists of two important tasks: unit selection and unit assignment. Unit selection determines the number and types of RT components to be used in the design. Unit assignment involves the mapping of the variables and operations in the scheduled control- and dataflow graph into the functional, storage and interconnection units.

In this paper we describe the basic methods that can be used to solve the allocation problem: greedy methods, which progressively construct a design while traversing the control- and dataflow graph; decomposition methods, which decompose the allocation problem into its constituent parts and solve each of them separately; iterative methods, which try to combine and interleave the solution of the allocation subproblems. We describe a greedy constructive algorithm, a decomposition method based on the graph-theory, called the clique partitioning algorithm, and an iterative refinement method, called the pairwise exchange algorithm.

Keywords: digital systems, high-level synthesis, datapath synthesis.

1. Introduction

In the process of high-level synthesis, the scheduling phase assigns operations to control steps, and thus converts a behavioral description into a set of register transfers that can be described by a state table. A target architecture for such a description is the *Finite State Machine with Datapath* (*FSMD*). The control unit for this model can be derived from the control-step sequence and the conditions used to determine the next control step in the sequence. The datapath is derived from the register transfers assigned to each control step; this task is called *datapath synthesis* or *datapath allocation*.

A datapath in the *FSMD* model is a netlist composed of three types of register transfer (RT) components: functional, storage and interconnection. The functional units (adders, shifters, ALUs) execute the operations specified in the behavioral description. The storage units (registers, RAMs, ROMs) hold the values of variables generated and consumed during the execution. The interconnection units (buses and multiplexers) transport data between the functional and storage units.

Datapath allocation consists of two important tasks: unit selection and unit assignment (binding). Unit selection determines the number and types of RT components to be used in the design. Unit assignment involves the mapping of the variables and operations in the scheduled control- and dataflow graph (CDFG) into the functional, storage and interconnection units, ensuring that the design behavior operates correctly on the selected sets of components. For every operation in the CFDG, we need a functional unit that can execute the operation. For every variable that is used across several control steps in the scheduled CDFG, we need a storage unit to hold the data values during the variable's lifetime. Finally, for every data transfer in the CDFG, we need a set of interconnection units to effect the transfer.

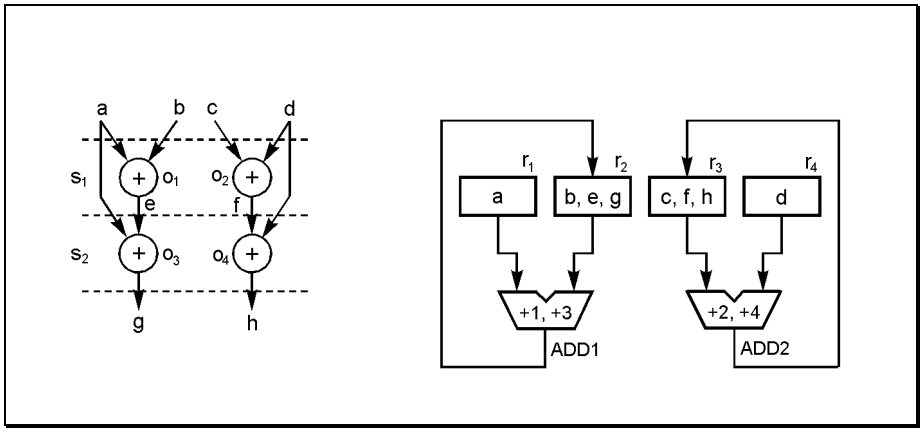

Figure 1. Mapping of behavioral objects into RT components.

The mapping of variables and operations in a dataflow graph (DFG) into RT components is illustrated in **Figure 1** [1]. We select two adders, *ADD1* and *ADD2*, and four registers, r_1, r_2, r_3 and r_4. Operations o_1 and o_2 cannot be mapped to the same adder, because they must be performed in the same control step s_1. On the other hand, operation o_1 can share an adder with operation o_3, because they are carried out during different control steps. Operations o_1 and o_3 are both mapped into *ADD1*. Variables *a* and *e* must be stored separately because their values are needed concurrently in control step s_2. Registers r_1 and r_2 must be connected to the input ports of *ADD1*; otherwise, operation o_3 will not be able to execute in *ADD1*. Similarly, operations o_2 and o_4 are mapped to *ADD2*. There are several different ways of performing the assignment. For example, we can map o_2 and o_3 to *ADD1*, and o_1 and o_4 to *ADD2*.

The allocation problem can be solved in three different ways: greedy methods, which progressively construct a design while traversing the CDFG; decomposition methods, which decompose the allocation problem into its constituent parts and solve each of them separately; iterative methods, which try to combine and interleave the solution of the allocation subproblems.

2. Greedy constructive methods

A constructive method starts with an empty datapath and builds the datapath gradually by adding functional, storage and interconnection units [2]. For each operation, the method tries to find a functional unit on the partially designed datapath that is capable of executing the operation and is idle during the control step in which the operation must be executed. If there are two or more functional units that meet these conditions, we chose the one which results in a minimal increase in the interconnection cost. On the other hand, if none of the functional units on the partially designed datapath meet the conditions, we add a new functional unit from the component library that is capable of carrying out the operation.

Similarly, we can assign a variable to an available register only if its lifetime interval does not overlap with those of variables already assigned to that register. A new register is allocated only when no allocated register meets the above condition. When multiple alternatives exist for assignment of a variable to a register, we select the one that minimally increases the datapath cost.

The greedy constructive allocation method is described in **Figure 2**. Let *UBE* be the set of unallocated behavioral entities and $DP_{current}$ be the partially designed datapath. The behavioral entities being considered could be variables that have to be mapped into registers, operations that have to be mapped into functional units, or data transfers that have to be mapped into interconnection units. Initially, $DP_{current}$ is empty. The procedure *ADD* (*DP*, *ube*) structurally modifies the datapath *DP* by adding to it the components necessary to support the behavioral entity *ube*. The function *COST* (*DP*) evaluates the area/performance cost of a partially designed datapath *DP*. DP_{temp} is a temporary datapath which is created in order to evaluate the cost c_{temp} of performing each modification to $DP_{current}$.

```
DP_current = φ;
while UBE ≠ φ do
    LowerCost = ∞;
    for all ube ∈ UBE do
        DP_temp = ADD (DP_current, ube);
        c_temp = COST (DP_temp);
        if c_temp < LowerCost then
            LowerCost = c_temp;
            BestEntity = ube;
        endif
    endfor
    DP_current = ADD (DP_current, BestEntity);
    UBE = UBE - BestEntity;
endwhile
```

Figure 2. The constructive allocation algorithm.

Starting with the set *UBE*, the *for* loop determines which unallocated behavioral entity, *BestEntity*, requires the minimal increase in the cost when added to the datapath. This is accomplished by adding each of the unallocated behavioral entities in *UBE* to $DP_{current}$ individually and then evaluating the resulting cost. The procedure *ADD* then modifies $DP_{current}$ by incorporating *BestEntity* to the datapath. *BestEntity* is deleted from the set of unallocated behavioral entities. The algorithm iterates in the *while* loop until all behavioral entities have been allocated (i.e., $UBE = \phi$).

In order to use the greedy constructive method, we have to address two basic problems: the cost-function calculation and the order in which the unallocated behavioral entities are mapped into the datapath. The costs can be computed as described in [1]. The order in which unallocated entities are mapped into the datapath can be determined either statically or dinamically. In a static approach, the objects are ordered before the datapath construction begins. The ordering is not changed during the construction process. In a dynamic approach, no ordering is done in advance. To select an operation or variable for assigning to the datapath, we evaluate every unallocated behavioral entity in terms of the cost involved in modifying the partial datapath, and the entity that requires the least expensive modification is chosen. After each assignment, we reevaluate the costs associated with the remaining unbound entities. The algorithm described uses the dynamic strategy.

The constructive method falls into the category of greedy algorithms. Although greedy algorithms are simple, the solutions they find can be far from optimal.

3. Decomposition methods

In order to improve the quality of the allocation results, we can use a decomposition method, where the allocation process is divided into a sequence of independent tasks; each task is transformed into a well-defined problem in graph theory and then solved with a proven technique.

While a greedy constructive method might interleave the storage, functional-unit, and interconnection allocation steps, decomposition methods will complete one task before performing another. Because of interdependencies among these tasks, no optimal solution is guaranteed even if all the tasks are solved optimally.

We describe one allocation technique based on the clique partitioning method from the graph-theory. The three tasks of storage, functional-unit and interconnection allocation can be solved independently by mapping each task to the problem of graph clique-partitioning [3].

```
/* create a super-graph G'(S, E'); */
S = ϕ; E' = ϕ;
for each v_i ∈ V do  s_i = {v_i}; S = S ∪ {s_i}; endfor
for each e_i,j ∈ E do  E' = E' ∪ {e'_i,j}; endfor
while E' ≠ ϕ do
    /* find s_Index1, s_Index2 having most common neighbors */
    MaxCommons = -1;
    for each e'_i,j ∈ E' do
        c_i,j = | COMMON_NEIGHBOR (G', s_i, s_j) |;
        if c_i,j > MaxCommons then
            MaxCommons = c_i,j;
            Index1 = i; Index2 = j;
        endif
    endfor
    CommonSet = COMMON_NEIGHBOR (G', s_Index1, s_Index2);
    /* delete all edges linking s_Index1 or s_Index2 */
    E' = DELETE_EDGE (E', s_Index1);
    E' = DELETE_EDGE (E', s_Index2);
    /* merge s_Index1 and s_Index2 into s_Index1Index2 */
    s_Index1Index2 = s_Index1 ∪ s_Index2;
    S = S - s_Index1 - s_Index2;
    S = S ∪ {s_Index1Index2};
    /* add edge from s_Index1Index2 to super-nodes in CommonSet */
    for each s_i ∈ CommonSet do
        E' = E' ∪ {e'_i,Index1Index2};
    endfor
endwhile
```

Figure 3. The clique partitionig algorithm.

Let $G = (V, E)$ a graph, where V is the set of vertices and E the set of edges. Each edge $e_{i,j} \in E$ links two different vertices v_i and $v_j \in V$. A subgraph SG of G is defined as (SV, SE), where $SV \subseteq V$ and $SE = \{e_{i,j} \mid e_{i,j} \in E, v_i, v_j \in SV\}$. A graph is complete if and only if for every pair of its vertices there exists an edge linking them. A clique of G is a complete subgraph of G. The problem of partitioning a graph into a minimal number of cliques such that

each node belongs to exactly one clique is called clique partitioning. For this problem heuristic procedures are usually used.

The algorithm in **Figure 3** describes a heuristic proposed by *Tseng* and *Siewiorek* [3] to solve the clique partitioning problem. A super-graph $G'(S, E')$ is derived from the original graph $G(V, E)$. Each node $s_i \in S$ is a super-node that can contain a set of one or more vertices $v_i \in V$. E' is identical with E except that the edges in E' link super-nodes in S. A super-node $s_i \in S$ is a common neighbor of the two super-nodes s_j and $s_k \in S$ if there exists edges $e_{i,j}$ and $e_{i,k} \in E'$. The function *COMMON_NEIGHBOR* (G', s_i, s_j) returns the set of super-nodes that are common neighbors of s_i and s_j in G'. The procedure *DELETE_EDGE*(E', s_i) deletes all edges in E' which have s_i as their end super-node.

Initially, each vertex $v_i \in V$ of G is placed in a separate super-node $s_i \in S$ of G'. At each step, the algorithm finds the super-nodes s_{Index1} and s_{Index2} in S such that they are connected by an edge and have the maximum number of common neighbors. The set *CommonSet* contains all the common neighbors of s_{Index1} and s_{Index2}. All edges originating from s_{Index1} or s_{Index2} in G' are deleted. These two super-nodes are merged into a single super-node, $s_{Index1Index2}$, which contains all the vertices of s_{Index1} and s_{Index2}. New edges are added from $s_{Index1Index2}$ to all the super-nodes in *CommonSet*. The above steps are repeated until there are no edges left in the graph. The vertices contained in each super-node $s_i \in S$ form a clique of the graph G.

Figure 4 illustrates the above algorithm. In the graph of **Figure 4(a)**, $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{e_{1,3}, e_{1,4}, e_{2,3}, e_{2,5}, e_{3,4}, e_{4,5}\}$. Initially, each vertex is placed in a separate super-node (s_1 .. s_5 in **Figure 4(b)**). The edges $e'_{1,3}$, $e'_{1,4}$ and $e'_{3,4}$ of the super-graph G' have the maximum number of common neighbors among all edges (**Figure 4(b)**). The first edge, $e'_{1,3}$, is selected and the following steps are carried out to yield the graph of **Figure 4(c)**:

1. s_4, the only common neighbor of s_1 şi s_3 is put in *CommonSet*.
2. All edges are deleted that link either super-nodes s_1 or s_3 to other super-nodes (i.e., $e'_{1,3}$, $e'_{1,4}$, $e'_{2,3}$ and $e'_{3,4}$).
3. Super-nodes s_1 and s_3 are combined into a new super-node s_{13}.
4. An edge is added between s_{13} and each super-node in *CommonSet*; i.e., the edge $e_{13,4}$ is added.

On the next iteration, s_4 is merged into s_{13} to yield the super-node s_{134} (**Figure 4(d)**). Finally, s_2 and s_5 are merged into the super-node s_{25} (**Figure 4(e)**). The cliques are $s_{134} = \{v_1, v_3, v_4\}$ and $s_{25} = \{v_2, v_5\}$ (**Figure 4(f)**).

In order to apply the clique partitioning technique to the allocation problem, we have to first derive the graph model from the input description. Consider as an example the register allocation. The primary goal of the register allocation is to minimize the register cost by maximizing the sharing of common registers among variables. To solve the register allocation problem, we construct a graph $G = (V, E)$, in which every vertex $v_i \in V$ uniquely represents a variable v_i and there exists an edge $e_{i,j} \in E$ if and only if variables v_i and v_j can be stored in the same register (i.e., their lifetime intervals do not overlap). All the variables whose representative vertices are in a clique of G can be stored in a single register. A clique partitioning of G provides a solution for the datapath storage-allocation problem that requires a minimum number of registers.

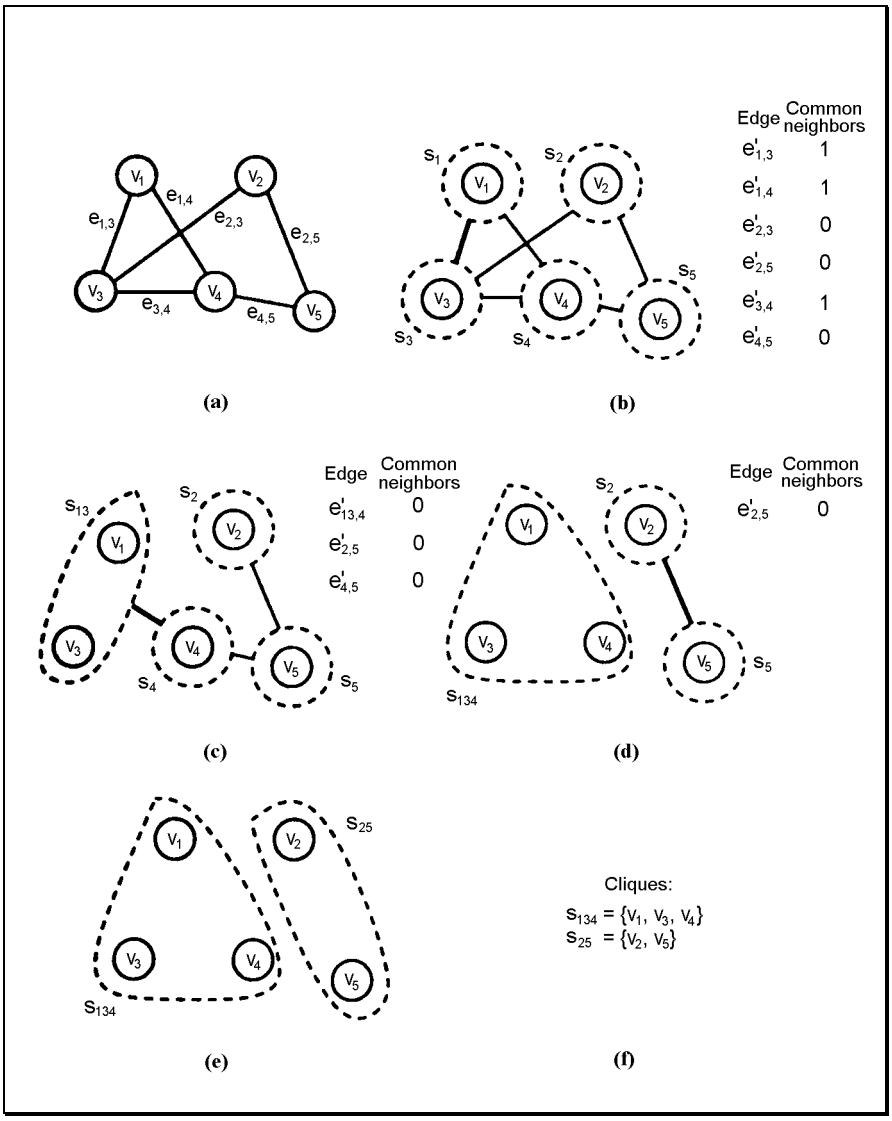

Figure 4. Clique partitioning.

Figure 5 shows a solution of the register-allocation problem using the clique partitioning algorithm.

Both functional-unit allocation and interconnection allocation can be formulated as a clique partitioning problem. For functional-unit allocation, each graph vertex represents an operation. An edge exists between two vertices if two conditions are satisfied:

1. the two operations are scheduled into different control steps, and
2. there exists a functional unit that is capable of carrying out both operations.

A clique-partitioning solution of this graph would yield a solution for the functional-unit allocation problem. Since a functional unit is assigned to each clique, all operations whose representative vertices are in a clique are executed in the same functional unit.

For interconnection-unit allocation, each vertex corresponds to a connection between two units. An edge links two vertices if the two corresponding connections are not used concurrently in any control step. A clique partitioning solution of such a graph implies partitioning of connections into buses or multiplexers. All connections whose representative vertices are in the same clique use the same bus or multiplexer.

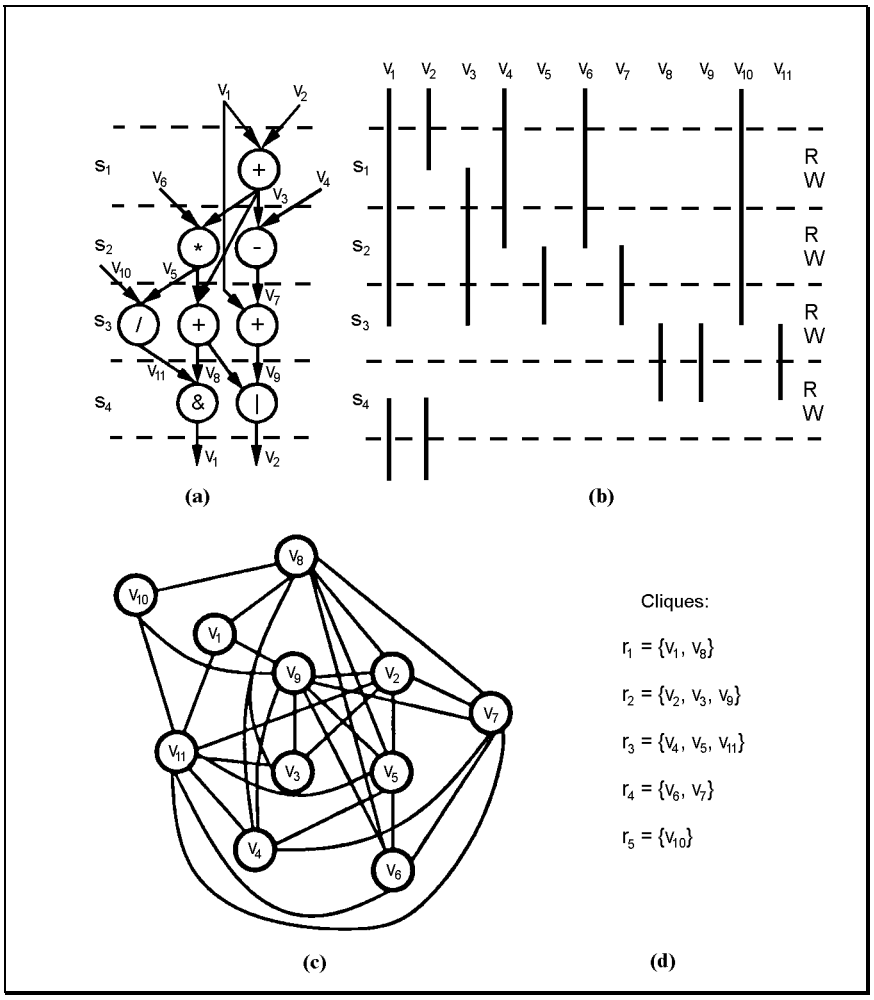

Figure 5. Register allocation using clique partitioning.

Although the clique partitioning method applied to storage allocation can minimize the storage requirements, it totally ignores the interdependence between storage and interconnection allocation. *Paulin* and *Knight* [4] extend the previous method by augmenting the graph edges with weights that reflect the impact on interconnection complexity due to register sharing among variables. An edge is given a higher weight if sharing of a register by the two variables corresponding to the edge's two end vertices reduces the inetrconnection cost. On the other hand, an edge is given a lower weight if the sharing causes an increase in the interconnection cost. The modified algorithm prefers cliques with heavier edges. Therefore, variables that share a common register are more likely to reduce the interconnection cost.

4. Iterative refinement methods

Given a datapath synthesized by constructive or decomposition methods, its quality can be improved by reallocation. Considering the functional-unit reallocation, the interconnection cost can be reduced by just swapping the functional-unit assignments for a pair of operations. Changing some variable to register assignments can be advantageous too.

The main issues in the iterative refinement are the types of modifications to be applied to a datapath, the selection of a modification type during an iteration and the termination criteria for the refinement process.

The most simple approach could be a simple assignment exchange. In this method, the modification to a datapath is limited to a swapping of two assignments (variable pairs or operation pairs). Assume that only operation swapping is used for the iterative refinement. The pairwise exchange algorithm performs a series of modifications to the datapath in order to decrease the cost of the datapath. First, all possible swappings of operation assignments scheduled into the same control step are evaluated in terms of the gain in the datapath cost due to a change in the interconnections. Then, the swapping that results in the largest gain is chosen and the datapath is updated to reflect the swapped operations. This process is repeated until no amount of swapping results in a positive gain (a futher reduction in the datapath cost).

The algorithm in **Figure 6** describes the pairwise exchange method [1]. Let $DP_{current}$ represent the current datapath structure and DP_{temp} represent a temporary datapath created to evaluate the cost of each operation assignment swap. The function *COST* (*DP*) evaluates the cost of the datapath *DP*. The datapath costs of $DP_{current}$ and DP_{temp} are represented by $c_{current}$ and c_{temp}. The procedure *SWAP* (*DP*, o_i, o_j) exchanges the assignments for operations o_i and o_j of the same type and updates the datapath *DP* accordingly. In each iteration of the innermost loop, *CurrentGain* represents the reduction in datapath cost due to the swapping of operations in that iteration. *BestGain* keeps track of the largest reduction in the cost attainable by any single swapping of operations evaluated so far in the current iteration.

```
repeat
    BestGain = –∞;
    c_current = COST (DP_current);
    for all control steps s do
        for each o_i, o_j of the same type scheduled into s, i ≠ j do
            DP_temp = SWAP (DP_current, o_i, o_j);
            c_temp = COST (DP_temp);
            CurrentGain = c_current - c_temp;
            if CurrentGain > BestGain then
                BestGain = CurrentGain;
                BestOp1 = o_i; BestOp2 = o_j;
            endif
        endfor
    endfor
    if BestGain > 0 then
        DP_current = SWAP (DP_current, BestOp1, BestOp2);
    endif
until BestGain ≤ 0
```

Figure 6. The pairwise exchange algorithm.

Suppose operation o_i has been assigned to functional unit fu_j and one of its input variables has been assigned to register r_k. The removal of operation o_i from fu_j will not eliminate the interconnection from r_k to fu_j unless no other operation that has been previously assigned to fu_j has its input variables assigned to r_k. The iterative refinement process has to approach the problem by considering multiple objects simultaneously. We must take into account the relationship between entities of different types. For example, the gain obtained in operation reallocation may be much higher if its input variables are also reallocated simultaneously.

The strategy of reallocating a group of different types of entities can be as simple as a greedy constructive algorithm or as sophisticated as a branch-and-bound search.

5. Conclusions

In this paper, we described the datapath allocation problem, which consists of four basic subtasks: unit selection, functional-unit assignment, storage assignment and interconnection assignment. We discussed the interdependencies among the subtasks that can be performed in an interleaved manner using a greedy constructive method, or sequentially, using a decomposition method. We also showed how to iteratively refine the datapath by a selective, controlled reallocation process.

The greedy constructive method is the simplest amongst all the approaches. It is easy to implement and computationally inexpensive, but is likely to produce inferior designs. The clique partitioning method solve the allocation tasks separately, and is applicable to storage, functional and interconnection unit allocation. The iterative refinement method achieves a high quality design at the expense of more computation time.

Future research in datapath allocation will need to improve the allocation algorithms in several directions. First, the allocation algorithms can be integrated with the scheduler in order to take advantage of the combination between scheduling and allocation. The number of control steps and the required number of functional units cannot accurately reflect the design quality. A fast allocator can quickly provide the scheduler with more information than just these two numbers. Consequently, the scheduler will be able to make more accurate decisions. Second, the algorithms must use more realistic cost functions based on physical design characteristics. Finally, allocation of more complex datapath structures must be incorporated. For example, the variables and arrays in the behavioral description could be partitioned into memories, a task that is complicated by the fact that memory accesses may take several clock cycles.

References

[1] D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin: *High-Level Synthesis. Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[2] K. Kucukcakar, A. C. Parker: *Data Path Tradeoffs using MABAL*. Proceedings of the 24th Design Automation Conference, pp. 210-215, 1990.

[3] C. J. Tseng, D. P. Siewiorek: *Automated Synthesis of Data Paths on Digital Systems*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 5, no. 3, pp. 379-395, July 1986.

[4] P. G. Paulin, J. P. Knight: *Force-Directed Scheduling for the Behavioral Synthesis of ASIC's*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, no. 6, pp. 661-679, June 1989.

[5] D. D. Gajski, F. Vahid, S. Narayan, J. Gong: *Specification and design of Embedded Systems*. P T R Prentice Hall, 1994.