the cost of other bindings done in the future.

8.5 Decomposition Approaches

The intuitive approach taken by the constructive method falls into the category of greedy algorithms. Although greedy algorithms are simple, the solutions they find can be far from optimal. In order to improve the quality of the results, some researchers have proposed a decomposition approach, 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 approach like the one described in Algorithm 8.1 might interleave the storage, functional-unit, and interconnection allocation steps, decomposition methods will complete one task before performing another. For example, all variable-to-register assignments might be completed before any operation-to-functional-unit assignments are performed, and vice versa. Because of interdependencies among these tasks, no optimal solution is guaranteed even if all the tasks are solved optimally. For example, a design using three adders may need one fewer multiplexer than the one using only two adders. Therefore, an allocation strategy that minimizes the usage of adders is justified only when an adder costs more than a multiplexer.

In this section we describe allocation techniques based on three graph-theoretical methods: clique partitioning, left-edge algorithm and the weighted bipartite matching algorithm. For the sake of simplicity, we illustrate these allocation techniques by applying them to behavioral descriptions consisting of straight-line code without any conditional branches.

8.5.1 Clique Partitioning

The three tasks of storage, functional-unit and interconnection allocation can be solved independently by mapping each task to the well known problem of graph clique-partitioning [TsSi86].

We begin by defining the clique-partitioning problem. Let $G = (V, E)$

denote 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. The clique-partitioning problem is a classic NP-complete problem for which heuristic procedures are usually used.

Algorithm 8.2 describes a heuristic proposed by Tseng and Siewiorek [TsSi86] 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 to E except that the edges in E' now 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 exist 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. These two super-nodes are merged into a single super-node, $s_{Index1Index2}$, which contains all the vertices of s_{Index1} and s_{Index2}. 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. 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 8.10 illustrates the above algorithm. In the graph of Figure 8.10(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 (labeled s_1 through s_5 in Figure 8.10(b)). The three 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 8.10(b)). The first edge, $e'_{1,3}$, is selected and the

Algorithm 8.2: Clique Partitioning.

/* create a super graph $G'(S, E')$ */
$S = \phi$; $E' = \phi$;
for each $v_i \in V$ **do** $s_i = \{v_i\}$; $S = S \cup \{s_i\}$; **endfor**
for each $e_{i,j} \in E$ **do** $E' = E' \cup \{e'_{i,j}\}$; **endfor**

while $E' \neq \phi$ **do**

/* find s_{Index1}, s_{Index2} having most common neighbors */
$MostCommons = -1$;
for each $e'_{i,j} \in E'$ **do**
$c_{i,j} = |$ COMMON_NEIGHBOR(G', s_i, s_j) $|$;
if $c_{i,j} > MostCommons$ **then**
$MostCommons = c_{i,j}$;
$Index1 = i$; $Index2 = j$;
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 $s_{Index1Index2}$ */
$s_{Index1Index2} = s_{Index1} \cup s_{Index2}$;
$S = S - s_{Index1} - s_{Index2}$;
$S = S \cup \{s_{Index1Index2}\}$;

/* add edge from $s_{Index1Index2}$ to super-nodes in $CommonSet$ */
for each $s_i \in CommonSet$ **do**
$E' = E' \cup \{e'_{i,Index1Index2}\}$;
endfor

endwhile

following steps are carried out to yield the graph of Figure 8.10(c).

(1) s_4, the only common neighbor of s_1 and s_3 is put in *CommonSet*.

(2) All edges are deleted that link either super-nodes s_1 or s_3 (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 8.10(d)). Finally, s_2 and s_5 are merged into the super-node s_{25} (Figure 8.10(e)). The cliques are $s_{134} = \{v_1, v_3, v_4\}$ and $s_{25} = \{v_2, v_5\}$ (Figure 8.10(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 register allocation as an example. The primary goal of 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 minimal number of registers. Figure 8.11 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.

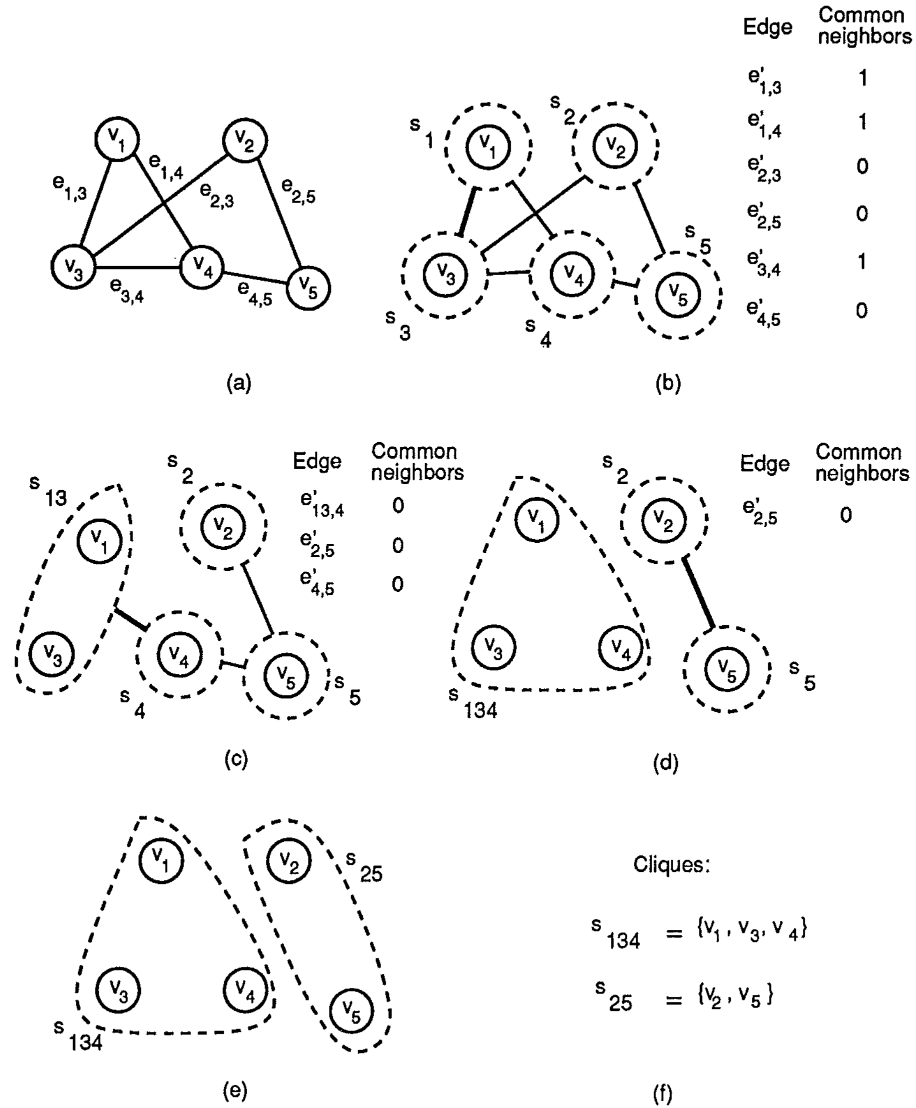

Figure 8.10: Clique partitioning: (a) given graph G, (b) calculating the common neighbors for the edges of graph G', (c) super-node s_{13} formed by considering edge $e'_{1,3}$, (d) super-node s_{134} formed by considering edge $e'_{13,4}$, (e) super-node s_{25} formed by considering edge $e'_{2,5}$, (f) resulting cliques s_{134} and s_{25}.

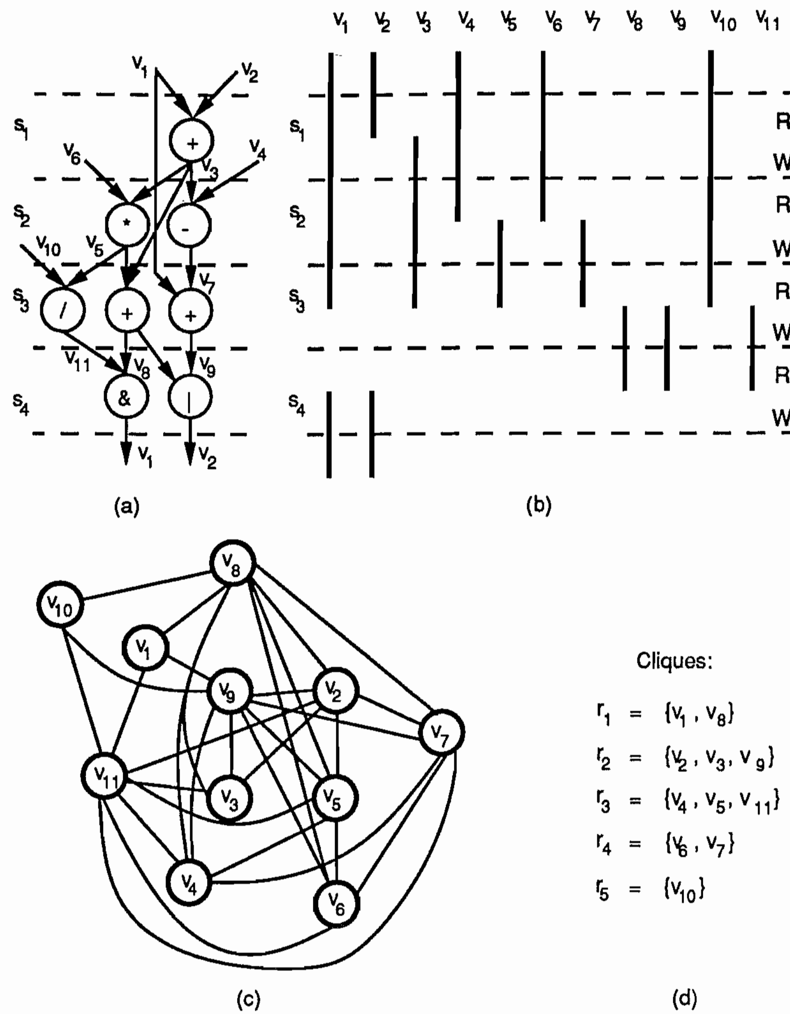

Figure 8.11: Register allocation using clique partitioning: (a) a scheduled DFG, (b) lifetime intervals of variables, (c) the graph model for register allocation, (c) a clique-partitioning solution.

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, whereas 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. In other words, all connections whose representative vertices are in the same clique use the same bus or multiplexer.

Although the clique-partitioning method when applied to storage allocation can minimize the storage requirements, it totally ignores the interdependence between storage and interconnection allocation. Paulin and Knight [PaKn89] 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 interconnection 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. Hence, variables that share a common register are more likely to reduce the interconnection cost.

8.5.2 Left-Edge Algorithm

The left-edge algorithm [HaSt71] is well known for its application in channel-routing tools for physical-design automation. The goal of the channel routing problem is to minimize the number of tracks used to connect points on the channel boundary. Two points on the channel boundary are connected with one horizontal (i.e., parallel to the channel) and two vertical (i.e., orthogonal to the channel) wire segments. Since the channel width depends on the number of horizontal tracks used, the channel-routing algorithms try to pack the horizontal segments into as few tracks as possible. Kurdahi and Parker [KuPa87] apply the left-edge algorithm to solve the register-allocation problem, in which variable lifetime intervals correspond to horizontal wire segments and registers to

wiring tracks.

The input to the left-edge algorithm is a list of variables, L. A lifetime interval is associated with each variable. The algorithm makes several passes over the list of variables until all variables have been assigned to registers. Essentially, the algorithm tries to pack the total lifetime of a new register allocated in each pass with as many variables whose lifetimes do not overlap, by using the channel-routing analogy of packing horizontal segments into as few tracks as possible.

Algorithm 8.3 describes register allocation using the left-edge algorithm. If there are n variables in the behavioral description, we define L to be the list of all variables v_i, $1 \leq i \leq n$. Let the 2-tuple $<Start(v), End(v)>$ represent the lifetime interval of a variable v where $Start(v)$ and $End(v)$ are respectively the start and end times of its lifetime interval. The procedure SORT(L) sorts the variables in L in ascending order with their start times, $Start(v)$, as the primary key and in descending order with their end times, $End(v)$, as the secondary key. The procedure DELETE (L, v) deletes the variable v from list L, FIRST(L) returns the first variable in the sorted list L and NEXT(L, v) returns the variable following v in list L. The array MAP keeps track of the registers assigned to each variable. The value of reg_index represents the index of the register being allocated in each pass. The end time of the interval of the most recently assigned variable in that pass is contained in $last$.

Initially, the variables are not assigned to any of the registers. During each pass over L, variables are assigned to a new register, r_{reg_index}. The first variable from the sorted list whose lifetime does not overlap with the lifetime of any other variables assigned to r_{reg_index} is assigned to the same register. When a variable is assigned to a register, the register is entered in the array MAP for that variable. On termination of the algorithm, the array MAP contains the registers assigned to all the variables and reg_index represents the total number of registers allocated.

Figure 8.12(a) depicts the sorted list of the lifetime intervals of the variables of the DFG in Figure 8.11(a). Note that variables v_1 and v_2 in Figure 8.11(a) are divided into two variables each (v_1, v_1' and v_2, v_2') in order to obtain a better packing density. Figure 8.12(b) illustrates how the left-edge algorithm works on the example. Starting with a new empty register r_1, the first variable in the sorted list, v_1, is put into r_1. Traveling

Algorithm 8.3: Register Allocation using Left-Edge Algorithm.

```
for all v ∈ L do   MAP[v] = 0;   endfor
SORT(L);

reg_index = 0;
while L ≠ φ do
   reg_index = reg_index + 1;
   curr_var = FIRST(L);
   last = 0;

   while curr_var ≠ null do
      if Start(curr_var) ≥ last then
         MAP[curr_var] = r_reg_index;
         last = End(curr_var);

         temp_var = curr_var;
         curr_var = NEXT(L, curr_var);
         DELETE(L, temp_var);
      else
         curr_var = NEXT(L, curr_var);
      endif
   endwhile

endwhile
```

down the list, no variables can be packed into r_1 before v_8 is encountered. After packing v_8 into r_1, the next packable variable down the list is $v_1{}'$. No more variables can be assigned to r_1 without overlapping variable lifetimes. Hence the algorithm allocates a new register (r_2) and starts from the beginning of the list again. The sorted list now has three fewer variables than it had in the beginning (i.e., v_1, v_8 and $v_1{}'$ have been removed). The list becomes empty after five registers have been allocated.

Unlike the clique-partitioning problem, which is NP-complete, the left-edge algorithm has a polynomial time complexity. Moreover, this algorithm allocates the minimum number of registers [KuPa87]. However, it cannot take into account the impact of register allocation on the interconnection cost, as can the weighted version of the clique-partitioning

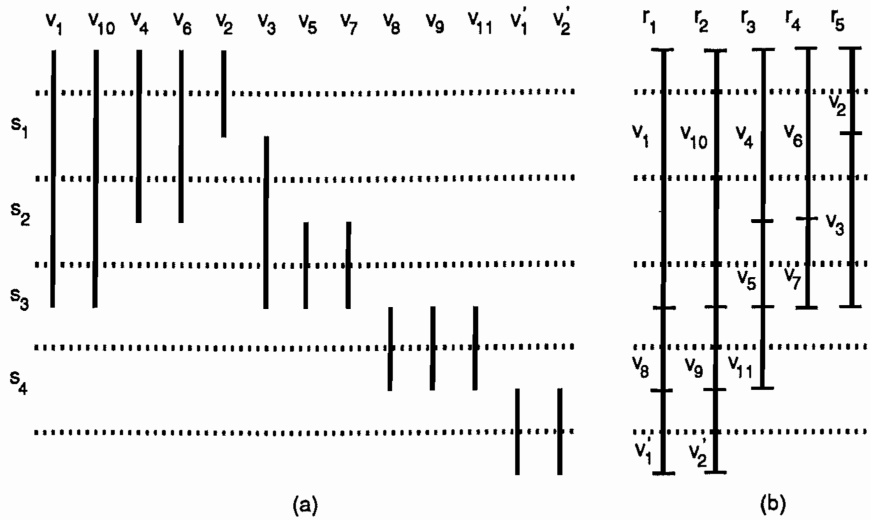

Figure 8.12: Register allocation using the left-edge algorithm: (a) sorted variable lifetime intervals, (b) five-register allocation result.

algorithm.

8.5.3 Weighted Bipartite-Matching Algorithm

Both the register and functional-unit allocation problems can be transformed into a weighted bipartite-matching algorithm [HCLH90]. Unlike the left-edge algorithm, which binds variables sequentially to one register at a time, the bipartite-matching algorithm binds multiple variables to multiple registers simultaneously. Moreover, it takes interconnection cost into account during allocation of registers and functional units.

Algorithm 8.4 describes the weighted bipartite-matching method. Let L be the list of all variables as defined in the description of the left-edge algorithm. The number of required registers, *num_reg*, is the maximum density of the lifetime intervals (i.e., the maximum number of overlapping variable lifetimes). Let R be the set of nodes representing