Chapter 5

Partitioning

Efficient designing of any complex system necessitates decomposition of the same into a set of smaller subsystems. Subsequently, each subsystem can be designed independently and simultaneously to speed up the design process. The process of decomposition is called *partitioning*. Partitioning efficiency can be enhanced within three broad parameters. First of all, the system must be decomposed carefully so that the original functionality of the system remains intact. Secondly, an interface specification is generated during the decomposition, which is used to connect all the subsystems. The system decomposition should ensure minimization of the interface interconnections between any two subsystems. Finally, the decomposition process should be simple and efficient so that the time required for the decomposition is a small fraction of the total design time.

Further partitioning may be required in the events where the size of a subsystem remains too large to be designed efficiently. Thus, partitioning can be used in a hierarchical manner until each subsystem created has a manageable size. Partitioning is a general technique and finds application in diverse areas. For example, in algorithm design, the *divide and conquer* approach is routinely used to partition complex problems into smaller and simpler problems. The increasing popularity of the parallel computation techniques brings in its fold promises in terms of provision of innovative tools for solution of complex problems, by combining partitioning and parallel processing techniques.

Partitioning plays a key role in the design of a computer system in general, and VLSI chips in particular. A computer system is comprised of tens of millions of transistors. It is partitioned into several smaller modules/blocks for facilitation of the design process. Each block has *terminals* located at the periphery that are used to connect the blocks. The connection is specified by a *netlist,* which is a collection of *nets.* A net is a set of terminals which have to be made electrically equivalent. Figure 5.1 (a) shows a circuit, which has been partitioned into three subcircuits. Note that the number of interconnections between any two partitions is four (as shown in Figure 5.1(b)).

A VLSI system is partitioned at several levels due to its complexity. At

the highest level, it is partitioned into a set of sub-systems whereby each subsystem can be designed and fabricated independently on a single PCB. High performance systems use MCMs instead of PCBs. At this level, the criterion for partitioning is the functionality and each PCB serves a specific task within a system. Consequently, a system consists of I/O (input /output) boards, memory boards, mother board (which hosts the microprocessor and its associated circuitry), and networking boards. Partitioning of a system into PCBs enhances the design efficiency of individual PCBs. Due to clear definition of the interface specified by the net list between the subsystems, all the PCBs can be designed simultaneously. Hence, significantly speeding up the design process.

If the circuit assigned to a PCB remains too large to be fabricated as a single unit, it is further partitioned into subcircuits such that each subcircuit can be fabricated as a VLSI chip. However, the layout process can be simplified and expedited by partitioning the circuit assigned to a chip into even smaller subcircuits. The partitioning process of a process into PCBs and an PCB into VLSI chips is physical in nature. That is, this partitioning is mandated by the limitations of fabrication process. In contrast, the partitioning of the circuit on a chip is carried out to reduce the computational complexity arising due to the sheer number of components on the chip. The hierarchical partitioning of a computer system is shown in Figure 5.2.

The partitioning of a system into a group of PCBs is called the *system level* partitioning. The partitioning of a PCB into chips is called the *board level* partitioning while the partitioning of a chip into smaller subcircuits is called the *chip level* partitioning. At each level, the constraints and objectives of the partitioning process are different as discussed below.

- **System Level Partitioning**: The circuit assigned to a PCB must satisfy certain constraints. Each PCB usually has a fixed area, and a fixed number of terminals to connect with other boards. The number of terminals available in one board (component) to connect to other boards (components) is called the *terminal count* of the board (component). For example, a typical board has dimensions 32 cm×15 cm and its terminal count is 64. Therefore, the subcircuit allocated to a board must be manufacturable within the dimensions of the board. In addition, the number of nets used to connect this board to the other boards must be within the terminal count of the board.

  The reliability of the system is inversely proportional to the number of boards in the system. Hence, one of the objectives of partitioning is to minimize the number of boards. Another important objective is the optimization of the system performance. Partitioning must minimize any degradation of the performance caused by the delay due to the connections between components on different boards. The signal carried by a net that is cut by partitioning at this level has to travel from one board to another board through the system bus. The system bus is very slow as the bus has to adhere to some strict specifications so that a variety

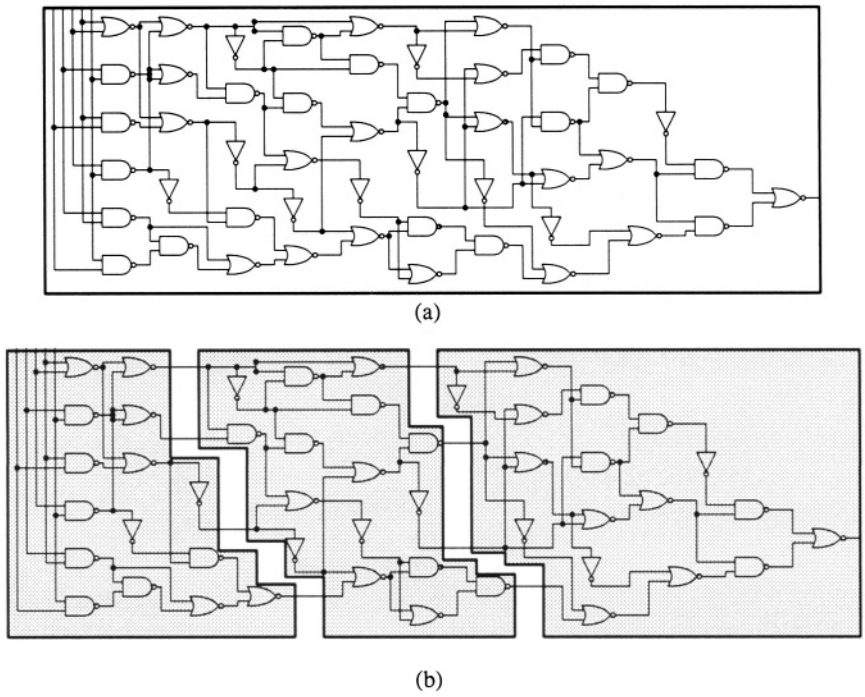


Figure 5.1: Partitioning of a circuit.

of different boards can share the same bus. The delay caused by signals traveling between PCBs (off-board delay) plays a major role in determining the system performance as this delay is much larger than the on-board or the on-chip delay.

- **Board Level Partitioning**: The board level partitioning faces a different set of constraints and fulfills a different set of objectives as opposed to system level partitioning. Unlike boards, chips can have different sizes and can accommodate different number of terminals. Typically the dimensions of a chip range from 2 mm×2 mm to 25 mm×25 mm. The terminal count of a chip depends on the package of the chip. A Dual In-line Package (DIP) allows only 64 pins while a Pin Grid Array (PGA) package may allow as many as 300 pins.

  While system level partitioning is geared towards satisfying the area and the terminal constraints of each partition, board level partitioning ventures to minimize the area of each chip. The shift of emphasis is attributable to the cost of manufacturing a chip that is proportional to its area. In addition, it is expedient that the number of chips used for each board be minimized for enhanced board reliability. Minimization of the

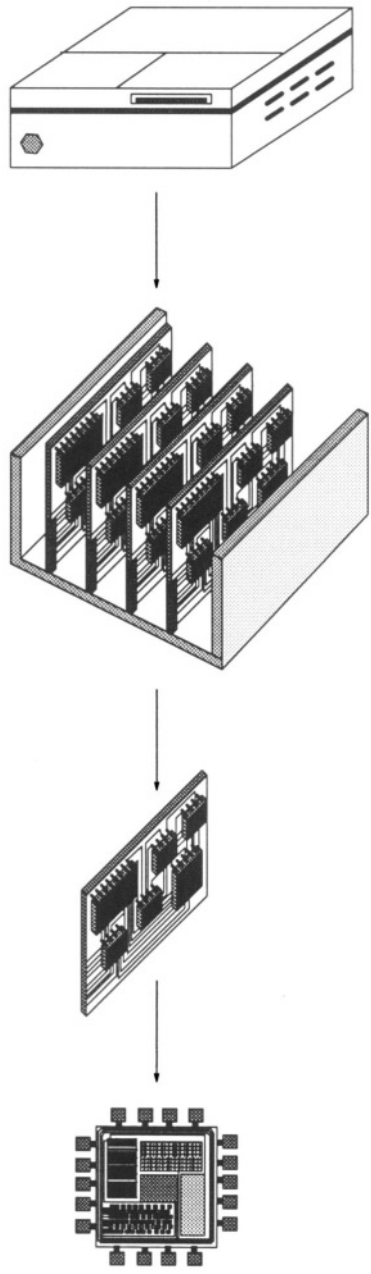


Figure 5.2: System hierarchy.

number of chips is another important determinant of performance because the off-chip delay is much larger than the on-chip delay. This differential in delay arises because the distance between two adjacent transistors on a chip is a few μm while the distance between two adjacent chips is in mm. In addition to traversing a longer distance, the signal has to travel between chips, and through the connector. The connector used to attach the chip to the board typically has a high resistance and contributes significantly to the signal delay. Figure 5.3 shows the different kinds of delay in a computer system. In Figure 5.3(b), the off-board delay is compared with the on-board delay while the off-chip delay is compared with the on-chip delay in Figure 5.3(c).

- **Chip Level Partitioning**: The circuit assigned to a chip can be fabricated as a single unit, therefore, partitioning at this level is necessary. A chip can accommodate as many as three million or more transistors. The fundamental objective of chip level partitioning is to facilitate efficient design of the chip.

  After partitioning, each subcircuit, which is also called a *block,* can be designed independently using either full custom or standard cell design style. Since partitioning is not constrained by physical dimensions, there is no area constraint for any partition. However, the partitions may be restrained by user specified area constraints for optimization of the design process.

  The terminal count for a partition is given by the ratio of the perimeter of the partition to the terminal pitch. The minimum spacing between two adjacent terminals is called *terminal pitch* and is determined by the design rules. The number of nets which connect a partition to other partitions cannot be greater than the terminal count of the partition. In addition, the number of nets cut by partitioning should be minimized to simplify the routing task. The minimization of the number of nets cut by partitioning is one of the most important objectives in partitioning.

  A disadvantage of the partitioning process is that it may degrade the performance of the final design. Figure 5.4(a) shows two components A and B which are critical to the chip performance, and therefore, must be placed close together. However, due to partitioning, components A and B may be assigned to different partitions and may appear in the final layout as shown in Figure 5.4(b). It is easy to see that the connection between A and B is very long, leading to a very large delay and degraded performance. Thus, during partitioning, these critical components should be assigned to the same partition. If such an assignment is not possible, then appropriate timing constraints must be generated to keep the two critical components close together. Chip performance is determined by several components forming a critical path. Assignment of these components to different partitions extends the length of the critical path. Thus, a major challenge for improvement of system performance is minimization of the length of critical path.

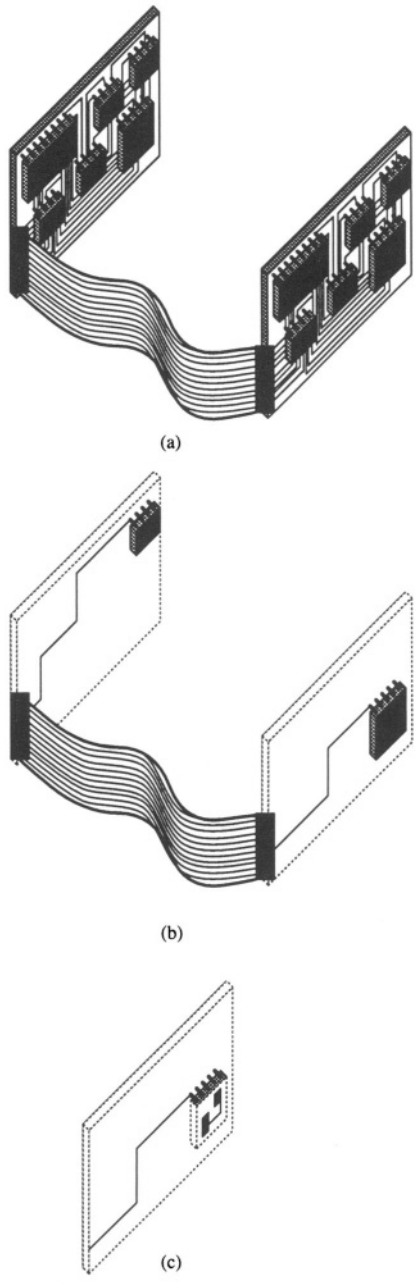


Figure 5.3: Different delays in a computer system.

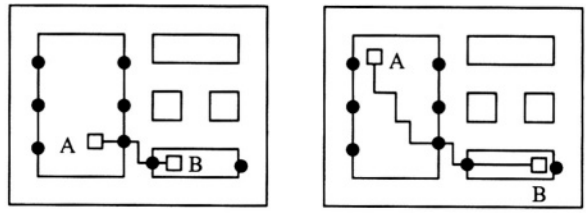


Figure 5.4: Bad partitioning increases the delay of circuit.

After a chip has been partitioned, each of the subcircuits has to be placed on a fixed plane and the nets between all the partitions have to be interconnected. The placement of the subcircuits is done by the placement algorithms and the nets are routed by using routing algorithms.

At any level of partitioning, the input to the partitioning algorithm is a set of components and a netlist. The output is a set of subcircuits which when connected, function as the original circuit and terminals required for each subcircuit to connect it to the other subcircuits. In addition to maintaining the original functionality, partitioning process optimizes certain parameters subject to certain constraints. The constraints for the partitioning problem include area constraints and terminal constraints. The objective functions for a partitioning problem include the minimization of the number of nets that cross the partition boundaries, and the minimization of the maximum number of times a path crosses the partition boundaries. The constraints and the objective functions used in the partitioning problem vary depending upon the partitioning level and the design style used. The actual objective function and constraints chosen for the partitioning problem may also depend on the specific problem.

5.1 Problem Formulation

The partitioning problem can be expressed more naturally in graph theoretic terms. A hypergraph $G = (V, E)$ representing a partitioning problem can be constructed as follows. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of vertices and $E = \{e_1, e_2, \ldots, e_m\}$ be a set of *hyperedges.* Each vertex represents a component. There is a hyperedge joining the vertices whenever the components corresponding to these vertices are to be connected. Thus, each hyperedge is a subset of the vertex set i.e., $e_i \subseteq V$, $i = 1, 2, \ldots, m$. In other words, each net is represented by a hyperedge. The area of each component is denoted as $a(v_i), 1 \leq i \leq n$. The modeling of partitioning problem into hypergraphs allows us to represent the circuit partitioning problem completely as a hypergraph partitioning problem. The partitioning problem is to partition V into

$V_1, V_2, \ldots, V_k$, where

$$V_i \cap V_j = \phi, \quad i \neq j$$
$$\cup_{i=1}^{k} V_i = V$$

Partition is also referred to as a *cut.* The cost of partition is called the *cut-size,* which is the number of hyperedges crossing the cut. Let c_{ij} be the cut-size between partitions V_i and V_j. Each partition V_i has an area $Area(V_i) = \sum_{v \in V_i} a(v)$, and a terminal count $Count(V_i)$. The maximum and the minimum areas, that a partition V_i can occupy, are denoted as $A_i^{\max}$ and $A_i^{\min}$, respectively. The maximum number of terminals that a partition V_i can have is denoted as T_i. Let $P = \{p_1, p_2, \ldots, p_m\}$ be a set of hyperpaths. Let $H(p_i)$ be the number of times a hyperpath p_i is cut, and let $K_{\min}$ and $K_{\max}$ represent the minimum and the maximum number of partitions that are allowed for a given subcircuit.

The constraints and the objective functions for the partitioning algorithms vary for each level of partitioning and each of the different design styles used. This makes it very difficult to state a general partitioning problem which is applicable to all levels of partitioning or all design styles used. Hence in this section we will list all the constraints and the objective functions and the level to which they are applicable. The partitioning problem at any level or design style deals with one or more of the following parameters.

1. **Interconnections between partitions:** The number of interconnections at any level of partitioning have to be minimized. Reducing the interconnections not only reduces the delay but also reduces the interface between the partitions making it easier for independent design and fabrication. A large number of interconnections increase the design area as well as complicate the task of the placement and routing algorithms. Minimization of the number of interconnections between partitions is called the *mincut* problem. The minimization of the cut is a very important objective function for partitioning algorithms for any level or any style of design. This function can be stated as:

$$Obj_1 : \sum_{i=1}^{k} \sum_{j=1}^{k} c_{ij}, (i \neq j) \quad \text{is minimized}$$

2. **Delay due to partitioning:** The partitioning of a circuit might cause a critical path to go in between partitions a number of times. As the delay between partitions is significantly larger than the delay within a partition, this is an important factor which has to be considered while partitioning high performance circuits. This is an objective function for partitioning algorithms for all levels of design. This objective function can be stated mathematically as:

$$Obj_2 : \max_{p_i \in P}(H(p_i)) \quad \text{is minimized}$$

3. **Number of terminals:** Partitioning algorithms at any level must partition the circuit so that the number of nets required to connect a subcircuit to other subcircuits does not exceed the terminal count of the subcircuit. In case of system level partitioning, this limit is decided by the maximum number of terminals available on a PCB connector which connects the PCB to the system bus. In case of board level partitioning, this limit is decided by the pin count of the package used for the chips. In case of chip level partitioning, the number of terminals of a subcircuit is determined by the perimeter of the area used by the subcircuit. At any level, the number of terminals for a partition is a constraint for the partitioning algorithm and can be stated as:

$$Cons_1 : Count(V_i) \leq T_i, \quad 1 \leq i \leq k$$

4. **Area of each partition:** In case of system level partitioning, the area of each partition (board) is fixed and hence this factor appears as a constraint for the system level partitioning problem. In case of board level partitioning, although it is important to reduce the area of each partition (chip) to a minimum to reduce the cost of fabrication, there is also an upper bound on the area of a chip, Hence, in this case also, the area appears as a constraint for the partitioning problem. At chip level, the size of each partition is not so important as long as the partitions are balanced. The area constraint can be stated as:

$$Cons_2 : A_i^{\min} \leq Area(V_i) \leq A_i^{\max}, \quad i = 1, 2, \ldots, k$$

5. **Number of partitions:** The number of partitions appears as a constraint in the partitioning problem at system level and board level partitioning. This prevents a system from having too many PCBs and a PCB from having too many chips. A large number of partitions may ease the design of individual partitions but they may also increase the cost of fabrication and the number of interconnections between the partitions. At the same time, if the number of partitions is small, the design of these partitions might still be too complex to be handled efficiently. At chip level, the number of partitions is determined, in part, by the capability of the placement algorithm. The constraint on the number of partitions can be stated as,

$$Cons_3 : K_{\min} \leq k \leq K_{\max}$$

Multiway partitioning is normally reduced to a series of two-way or *bipartitioning* problem. Each component is hierarchically bipartitioned until the desired number of components is achieved. In this chapter, we will restrict ourselves to bipartitioning. When the two partitions have the same size, the partitioning process is called *bisectioning* and the partitions are called *bisections.*

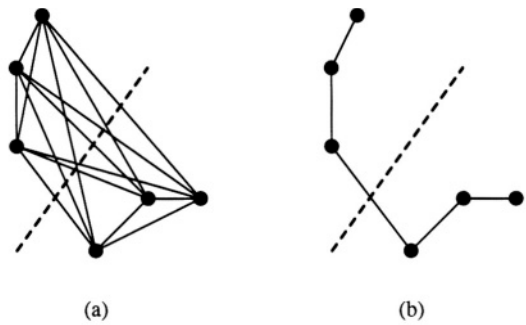


Figure 5.5: A net represented as a clique and a spanning tree.

An actual model representing the partitioning problem to be solved at system level or board level requires that the area constraint, interconnection constraint and constraint on the number of partitions be satisfied. Therefore, constraints $Cons_1, Cons_2$, and $Cons_3$ apply. If the performance of the system is also a criterion, then the objective function Obj_2 is also applicable. At chip level, the partitioning algorithms usually have Obj_1 as an objective function. In case of high performance circuits, objective function Obj_2 is also applicable.

An important factor, not discussed above, is modeling of a net. So far, we have assumed that a net is modeled as a hyperedge. However, hyperedges are hard to handle and the model is sometimes simplified. One way of simplifying the model is to represent each hyperedge by a clique of its vertices. However using this method increases the number of times the edges cross boundaries substantially as shown in Figure 5.5(a). There are other ways to represent hyperedges. For example, we can use a tree to represent a hyperedge as shown in Figure 5.5(b), but doing this destroys the symmetric property of the clique model. In general, net modeling is a hard problem and no satisfactory solution has been proposed.

5.1.1 Design Style Specific Partitioning Problems

The problems formulated above represent a general approach to partitioning. However, partitioning algorithms for different design styles have different objectives. In this section, we will discuss the partitioning problems for each design style. Partitioning problems for FPGAs and MCM will be discussed in Chapters 11 and 12, respectively.

1. **Full custom design style:** In a full custom design style, partitions can be of different sizes and hence there are no area constraints for the partitioning algorithms. Thus, the partitioning in full custom design style has the most flexibility. During chip level partitioning, the number of terminals allowed for each partition is determined by the perimeter

of the block corresponding to a partition. Thus, the estimated terminal count for a partition i is given by

$$T_i = \frac{p_i}{d}, \quad i = 1, 2, \ldots, k$$

where, p_i is the perimeter of the block corresponding to the partition i and d is the terminal pitch. Since, the cost of manufacturing a circuit is directly proportional to the layout size, it is essential to keep the area of the layout to a minimum. The area of circuit layout is the sum of the areas occupied by components, areas used for routing the nets, and the unused areas. Since the areas occupied by the components are fixed, it is only possible to minimize the routing areas and unused areas. The routing area will be largely used by the nets that go across the boundaries of the blocks. The amount of unused areas will be determined by the placement. Therefore in addition to the terminal constraints, partitioning algorithms have to minimize the total number of nets that cross the partition boundaries. A partitioning algorithm for full custom design has objective function Obj_1 subject to the constraints $Cons_1$ and $Cons_2$. The full custom design style is typically used for the design of high-performance circuits, e.g., design of microprocessors. The delay for high-performance circuits is of critical importance. Therefore, an additional objective function Obj_2 is added to the partitioning problem for the full custom design style.

2. **Standard cell design style:** The primary objective of the partitioning algorithms in standard cell design style is to partition the circuit into a set of disjoint subcircuits such that each subcircuit corresponds to a cell in a standard cell library. In addition, the partitioning procedure is non-hierarchical. The complexity of partitioning depends on the type of the standard cells available in the standard cell library. If the library has only a few simple cell types available, there are few options for the partitioning procedure and the partitioning problem has to satisfy constraints $Cons_1$ and $Cons_2$. However, if there are many cell types available, some of which are complex, then the partitioning problem is rather complicated. The objective function to be optimized by the partitioning algorithms for standard cell design is Obj_1. For high performance circuits, Obj_1 and Obj_2 are used as combined objective functions.

3. **Gate array design style:** The circuit is bipartitioned recursively until each resulting partition corresponds to a gate on the gate array. The objective for each bipartitioning is to minimize the number of nets that cross the partition boundaries.

In future VLSI chips, the terminals may be on top of the chip and therefore terminal counts have to be computed accordingly. In addition, due to ever-reducing routing areas, the transistors will get packed closer together and

thermal constraints may become dominant, as they are in MCM partitioning problems.

5.2 Classification of Partitioning Algorithms

The mincut problem is NP-complete, it follows that general partitioning problem is also NP-complete [GJ79]. As a result, variety of heuristic algorithms for partitioning have been developed. Partitioning algorithms can be classified in three ways. The first method of classification depends on availability of initial partitioning. There are two classes of partitioning algorithms under this classification scheme:

1. Constructive algorithms and
2. Iterative algorithms.

The input to a *constructive* algorithms is the circuit components and the netlist. The output is a set of partitions and the new netlist. Constructive algorithms are typically used to form some initial partitions which can be improved by using other algorithms. In that sense, constructive algorithms are used as preprocessing algorithms for partitioning. They are usually fast, but the partitions generated by these algorithms may be far from optimal.

Iterative algorithms, on the other hand, accept a set of partitions and the netlist as input and generate an improved set of partitions with the modified netlist. These algorithms iterate continuously until the partitions cannot be improved further.

The partitioning algorithms can also be classified based on the nature of the algorithms. There are two types of algorithms:

1. Deterministic algorithms and
2. Probabilistic algorithms.

Deterministic algorithms produce repeatable or deterministic solutions. For example, an algorithm which makes use of deterministic functions, will always generate the same solution for a given problem. On the other hand, the *probabilistic* algorithms are capable of producing a different solution for the same problem each time they are used, as they make use of some random functions.

The partitioning algorithms can also be classified on the basis of the process used for partitioning. Thus we have the following categories:

1. Group Migration algorithms,
2. Simulated Annealing and Evolution based algorithms and
3. Other partitioning algorithms.

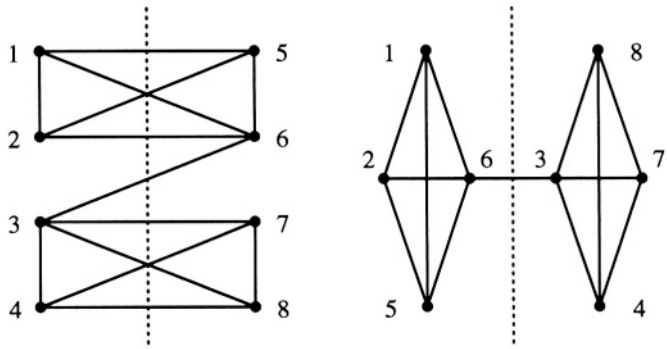
The *group migration algorithms* [FM82, KL70] start with some partitions, usually generated randomly, and then move components between partitions to improve the partitioning. The group migration algorithms are quite efficient. However, the number of partitions has to be specified which is usually not known when the partitioning process starts. In addition, the partitioning of an entire system is a multi-level operation and the evaluation of the partitions obtained by the partitioning depends on the final integration of partitions at all levels, from the basic subcircuits to the whole system. An algorithm used to find a minimum cut at one level may sacrifice the quality of cuts for the following levels. The group migration method is a deterministic method which is often trapped at a local optimum and can not proceed further.

The *simulated annealing/evolution* [CH90, GS84, KGV83, RVS84] algorithms carry out the partitioning process by using a cost function, which classifies any feasible solution, and a set of moves, which allows movement from solution to solution. Unlike deterministic algorithms, these algorithms accept moves which may adversely effect the solution. The algorithm starts with a random solution and as it progresses, the proportion of adverse moves decreases. These degenerate moves act as a safeguard against entrapment in local minima. These algorithms are computationally intensive as compared to group migration and other methods.

Among all the partitioning algorithms, the group migration and simulated annealing or evolution have been the most successful heuristics for partitioning problems. The use of both these types of algorithms is ubiquitous and extensive research has been carried out on them. The following sections include a detailed discussion of these algorithms. The remaining methods will be discussed briefly later in the chapter.

5.3 Group Migration Algorithms

The *group migration* algorithms belong to a class of iterative improvement algorithms. These algorithms start with some initial partitions, formed by using a constructive algorithm. Local changes are then applied to the partitions to reduce the cutsize. This process is repeated until no further improvement is possible. Kernighan and Lin (K-L) [KL70] proposed a graph bisectioning algorithm for a graph which starts with a random initial partition and then uses pairwise swapping of vertices between partitions, until no improvement is possible. Schweikert and Kernighan [SK72] proposed the use of a net model so that the algorithm can handle hypergraphs. Fiduccia and Mattheyses [FM82] reduced time complexity of K-L algorithm to $O(t)$, where t is the number of terminals. An algorithm using vertex-replication technique to reduce the number of nets that cross the partitions was presented by Kring and Newton [KN91]. Goldberg and Burstein [GB83] suggested an algorithm which improves upon the original K-L algorithm using graph matchings. One of the problems with the K-L algorithm is the requirement of prespecified sizes of partitions. Wei and Cheng [WC89] proposed a ratio-cut model in which the sizes of the par-



(a) Initial Bisections (b) Final Bisections

Figure 5.6: A graph bisected by K-L algorithm.

titions do not need to be specified. The algorithms based on group migration are used extensively in partitioning VLSI circuits. In the following sections, these algorithms are discussed in detail.

5.3.1 Kernighan-Lin Algorithm

The K-L algorithm is a bisectioning algorithm. It starts by initially partitioning the graph $G = (V, E)$ into two subsets of equal sizes. Vertex pairs are exchanged across the bisection if the exchange improves the cutsize. The above procedure is carried out iteratively until no further improvement can be achieved.

Let us illustrate the basic idea of the K-L algorithm with the help of an example before presenting the algorithm formally. Consider the example given in Figure 5.6(a). The initial partitions are

$$A = \{1, 2, 3, 4\}$$
$$B = \{5, 6, 7, 8\}$$

Notice that the initial cutsize is 9. The next step of K-L algorithm is to choose a pair of vertices whose exchange results in the largest decrease of the cutsize *or* results in the smallest increase, if no decrease is possible. The decrease of the cutsize is computed using gain values $D(i)$ of vertices v_i. The gain of a vertex v_i is defined as

$$D(i) = inedge(i) - outedge(i)$$

where $inedge(i)$ is the number of edges of vertex i that do not cross the bisection boundary and $outedge(i)$ is the number of edges that cross the boundary. The amount by which the cutsize decreases, if vertex v_i changes over to the other

partition, is represented by *D(i).* If v_i and v_j are exchanged, the decrease of cutsize is $D(i) + D(j)$. In the example given in Figure 5.6, a suitable vertex pair is (3, 5) which decreases the cutsize by 3. A tentative exchange of this pair is made. These two vertices are then locked. This lock on the vertices prohibits them from taking part in any further tentative exchanges. The above procedure is applied to the new partitions, which gives a second vertex pair of (4, 6). This procedure is continued until all the vertices are locked. During this process, a log of all tentative exchanges and the resulting cutsizes is stored. Table 5.1 shows the log of vertex exchanges for the given example. Note that the partial sum of cutsize decrease $g(i)$ over the exchanges of first i vertex pairs is given in the table e.g., $g(1) = 3$ and $g(2) = 8$. The value of k for which $g(k)$ gives the maximum value of all $g(i)$ is determined from the table. In this example, $k = 2$ and g(2) = 8 is the maximum partial sum. The first k pairs of vertices are actually exchanged. In the example, the first two vertex pairs (3, 5) and (4, 6) are actually exchanged, resulting in the bisection shown in Figure 5.6(b). This completes an iteration and a new iteration starts. However, if no decrease of cutsize is possible during an iteration, the algorithm stops. Figure 5.7 presents the formal description of the K-L algorithm.

The procedure INITIALIZE finds initial bisections and initializes the parameters in the algorithm. The procedure IMPROVE tests if any improvement has been made during the last iteration, while the procedure UNLOCK checks if any vertex is unlocked. Each vertex has a status of either *locked* or *unlocked.* Only those vertices whose status is *unlocked* are candidates for the next tentative exchanges. The procedure TENT-EXCHGE tentatively exchanges a pair of vertices. The procedure LOCK locks the vertex pair, while the procedure LOG stores the log table. The procedure ACTUAL-EXCHGE determines the maximum partial sum of $g(i)$, selects the vertex pairs to be exchanged and fulfills the actual exchange of these vertex pairs.

The time complexity of Kernighan-Lin algorithm is $O(n^3)$. The Kernighan-Lin algorithm is, however, quite robust. It can accommodate additional constraints, such as a group of vertices requiring to be in a specified partition. This feature is very important in layout because some blocks of the circuit are to be kept together due to the functionality. For example, it is important to keep all components of an adder together. However, there are several disadvantages of K-L algorithm. For example, the algorithm is not applicable for hypergraphs, it cannot handle arbitrarily weighted graphs and the partition sizes have to be specified before partitioning. Finally, the complexity of the algorithm is considered too high even for moderate size problems.

5.3.2 Extensions of Kernighan-Lin Algorithm

In order to overcome the disadvantages of Kernighan-Lin Algorithm, several algorithms have been developed. In the following, we discuss several extensions of K-L algorithm.

```
Algorithm KL
begin
    INITIALIZE();
    while( IMPROVE(table) = TRUE ) do
    (* if an improvement has been made during last iteration,
    the process is carried out again. *)
        while ( UNLOCK(A) = TRUE ) do
        (* if there exists any unlocked vertex in A,
        more tentative exchanges are carried out. *)
            for ( each a ∈ A ) do
                if (a = unlocked) then
                    for( each b ∈ B ) do
                        if (b = unlocked) then
                            if (D_max < D(a) + D(b)) then
                                D_max = D(a) + D(b);
                                a_max = a;
                                b_max = b;
            TENT-EXCHGE(a_max, b_max);
            LOCK(a_max, b_max);
            LOG(table);
            D_max = −∞;
        ACTUAL-EXCHGE(table);
end.
```

Figure 5.7: Algorithm K-L

i	Vertex Pair	$g(i)$	$\sum_{j=1}^{i} g(i)$	Cutsize
0	-	-	-	9
1	(3,5)	3	3	6
2	(4,6)	5	8	1
3	(1,7)	-6	2	7
4	(2,8)	-2	0	9

Table 5.1: The log of the vertex exchanges.

5.3.2.1 Fiduccia-Mattheyses Algorithm

Fiduccia and Mattheyses [FM82] developed a modified version of Kernighan-Lin algorithm. The first modification is that only a single vertex is moved across the cut in a single move. This permits the handling of unbalanced partitions and nonuniform vertex weights. The other modification is the extension of the concept of cutsize to hypergraphs. Finally, the vertices to be moved across the cut are selected in such a way so that the algorithm runs much faster. As in Kernighan-Lin algorithm, a vertex is locked when it is tentatively moved. When no moves are possible, only those moves which give the best cutsize are actually carried out.

The data structure used for choosing the next vertex to be moved is shown in Figure 5.8. Each component is represented as a vertex. The vertex (component) gain is an integer and each vertex has its gain in the range $-pmax$ to $+pmax$ where $pmax$ is the maximum vertex degree in the hypergraph. Since vertex gains have restricted values, 'bucket' sorting can be used to maintain a sorted list of vertex gains. This is done using an array BUCKET[-pmax, ..., pmax], whose kth entry contains a doubly-linked list of free vertices with gains currently equal to k. Two such arrays are needed, one for each block. Each array is maintained by moving a vertex to the appropriate bucket whenever its gain changes due to the movement of one of its neighbors. Direct access to each vertex, from a separate field in the VERTEX array, allows removal of a vertex from its current list and its movement to the head of its new bucket list in constant time. As only free vertices are allowed to move, therefore, only their gains are updated. Whenever a base vertex is moved, it is locked, removed from its bucket list, and placed on a FREE VERTEX LIST, which is later used to reinitialize the BUCKET array for the next pass. The FREE VERTEX LIST saves a great deal of work when a large number of vertices (components) have permanent block assignments and are thus not free to move. For each BUCKET array, a MAXGAIN index is maintained which is used to keep track of the bucket having a vertex of highest gain. This index is updated by decrementing it whenever its bucket is found to be empty and resetting it to a higher bucket whenever a vertex moves to a bucket above MAXGAIN. Experimental results on real circuits have shown that gains tend to cluster sharply around the origin and that MAXGAIN moves very little, making the above implementation exceptionally fast and simple.

The total run time taken to update the gain values in one pass of the above algorithm is $O(n)$, where n is the number of terminals in the graph G. The F-M algorithm is much faster than Kernighan-Lin algorithm. A significant weakness of F-M algorithm is that the gain models the effect of a vertex move upon the size of the net cutsize, but not upon the gain of the neighboring vertices. Thus the gain does not differentiate between moves that may increase the probability of finding a better partition by improving the gains of other vertices and moves that reduce the gains of neighboring vertices. Krishnamurthy [Kri84] has proposed an extension to the F-M algorithm that accounts for high-order gains to get better results and a lower dependence upon the initial partition.

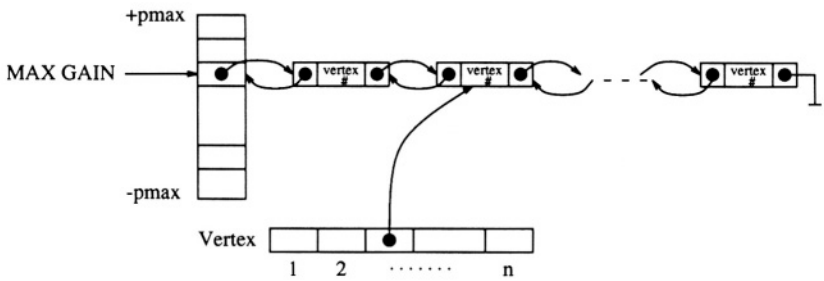


Figure 5.8: The data structure for choosing vertices.

5.3.2.2 Goldberg and Burstein Algorithm

Experimental results have shown that the quality of the final bisection obtained by iterative bisection algorithms, such as K-L algorithm, depends heavily on the ratio of the number of edges to the number of vertices [GB83]. The K-L algorithm yields good bisection if the ratio is higher than 5. However, if the ratio is less than 3, the algorithm performs poorly. The ratio in a typical problem of VLSI design is between 1.8 and 2.5. As a result, Goldberg and Burstein suggested an improvement to the original K-L algorithm or other bisection algorithms by using a technique of contracting edges to increase that ratio.

The basic idea of Goldberg-Burstein algorithm is to find a matching M in graph G, as shown in Figure 5.9(a). The thick lines indicate the edges which form matching. Each edge in the matching is contracted (and forms a vertex) to increase the density of graph. Contraction of edges in M is shown in Figure 5.9(b). Any bisection algorithm is applied to the modified graph and finally, edges are uncontracted within each partition.

5.3.2.3 Component Replication

Recall that the partitioning problem is to partition V into $V_1, V_2, \ldots, V_k$, such that

$$V_i \cap V_j = \phi, \quad i \neq j$$
$$\cup_{i=1}^{k} V_i = V$$

In component (vertex) replication technique, the condition that

$$V_i \cap V_j = \phi, \quad i \neq j$$

is dropped. That is, some vertices are allowed to be duplicated in two or more partitions. The vertex replication technique, presented by Kring and Newton [KN91], can substantially reduce the number of nets that cross boundaries of partitions. Figure 5.10(a) shows a partitioning of a circuit without vertex replication. However, when the inverters are replicated, as in Figure 5.10(b),

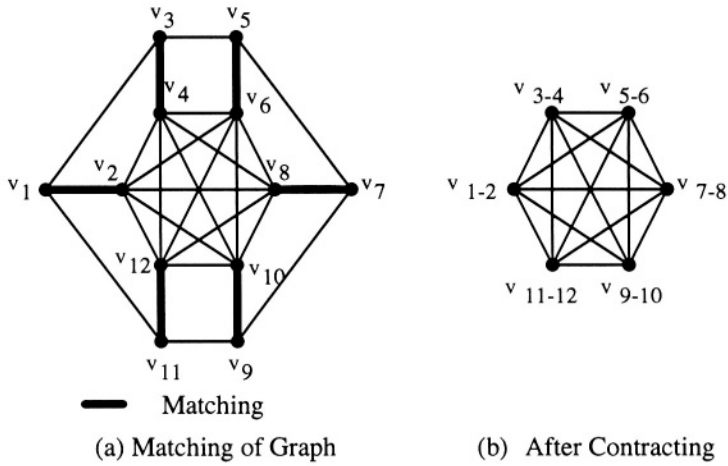


Figure 5.9: Matching and edge contraction in graph.

the cutsize is reduced. When a component is replicated, it is copied into both subcircuits and its output are generated locally and do not contribute to the cutsize. Replication does require the inputs to the component to be available on both sides of the partition. If inputs are not available on both sides, the inputs must be propagated across the partition and will contribute to the cutsize.

Once a vertex has been replicated, it tends to remain so and nets connected to the components remain in both subcircuits. Thus, while vertex replication does reduce the cutsize, it tends to reduce the ability to further improve the partition. To achieve good results with this technique, it is critical to limit component replication to where it is most useful by actively limiting the number of replicated components.

The replications of vertices must be done very carefully as in some situations, vertex replication may outweigh the benefit of a reduced cutsize. For example, the added redundancy may increase the circuit area, fault rate and testing. Also, vertex replication cannot be adopted by an arbitrary algorithm. Only those algorithms which carry out partitioning at component level can combine vertex replication techniques to reduce the cutsize. When vertex replication is used in algorithms which deal with more than one components at a time [Kri84], the vertex replication technique can actually increase the cutsize. However, there are cases, especially at the system level, where vertex replication is of great advantage. The algorithm has been tested for two types of circuits, combinational circuits and industrial circuits. The results are summarized in Table 5.2 in which the net cutsize reduction is the percentage reduction in the total number of partitioned nets when compared to partitions obtained without component replication. The component replication is the percentage of the total number of replicated components to the total number of compo-

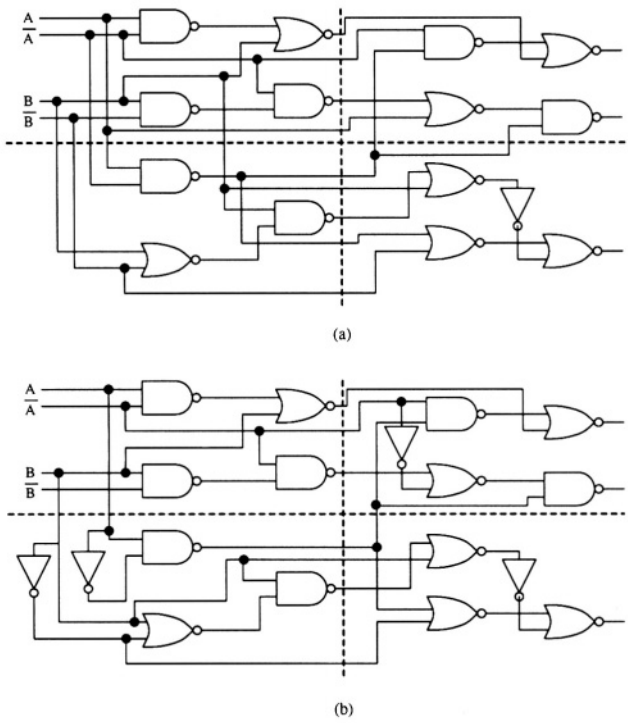


Figure 5.10: Component replication to reduce the cutsize.

nents. Table 5.2 clearly shows that vertex replication can substantially reduce the number of partitioned nets without significantly increasing the size of the circuit.

5.3.2.4 Ratio Cut

The Kernighan-Lin algorithm yields partitions of comparable sizes, but these sizes are predefined before partitioning. Since, there are natural clustering structures in the circuit, predefining the partition size may not be well suited for partitioning circuits, since there is no way to know the cluster size in circuits before partitioning. To remedy this situation, Wei and Cheng proposed the ratio cut as a new metric in order to locate natural clusters in the circuit and at the same time force the partitions to be of equal sizes [WC89]. Given a hypergraph $G = (V, E)$, let c_{ij} be the capacity of an edge connecting node i and node j. Let (V_1, V_2) be a cut that separates a set of nodes V_1 from its complement $V_2$where $V_2 = V - V_1$. The capacity of this cut is equal to $C_{V_1V_2} = \sum_{i \in V_1} \sum_{j \in V_2} c_{ij}$. The ratio of this cut $R_{V_1V_2}$ is defined as $R_{V_1V_2} = \frac{C_{V_1V_2}}{|V_1| \times |V_2|}$, where $|V_1|$ and $|V_2|$ denote the cardinalities of subsets V_1 and V_2 respectively.

Circuit Type	Maximum Circuit Expansion	Net Cutset Reduction	Component Replication
Combinational	10%	41%	3.2%
	30%	43%	3.9%
Industrial	10%	15%	0.7%
	30%	9%	0.3%

Table 5.2: Comparison of different design styles.

The ratio cut is the cut that generates the minimum ratio. The maximum flow minimum cut method [FF62] prefers very uneven subsets which naturally give the lowest cost. Instead of minimizing the cost $C_{V_1V_2}$, the ratio cut based approach minimizes the ratio $R_{V_1V_2}$ to alleviate this hidden size effect. Cuts that go through weakly connected groups and groups of similar sizes correspond to smaller ratios. In this way, the minimization of all cuts according to their corresponding ratios balances the effect of minimizing the cost and the effect of keeping the resulting partitions of similar sizes.

Like many other partitioning problems, finding the ratio cut in a hypergraph belongs to the class of NP-complete problems [MS86]. Therefore, a good and fast heuristic algorithm is needed. A heuristic based on Fiduccia and Mattheyses algorithm was proposed in [WC89].

5.4 Simulated Annealing and Evolution

Simulated annealing and evolution belong to the probabilistic and iterative class of algorithms. The simulated annealing algorithm for partitioning is the simulation of the annealing process used for metals. As in the actual annealing process, the value of temperature is decreased slowly till it approaches the freezing point. The simulated evolution algorithm, simulates the biological process of evolution. Each solution is called a *generation.* The generations are improved in each iteration by using operators which simulate the biological events in the evolution process.

5.4.1 Simulated Annealing

Simulated Annealing is a special class of randomized local search algorithms. The optimization of a circuit partitioning with a very large number of components is analogous to the process of annealing, in which a material is melted and cooled down so that it will crystallize into highly ordered state. The energy within the material corresponds to the partitioning score. In an annealing process, the solid-state material is heated to a high temperature until it reaches an amorphous liquid state. It is then cooled very slowly according to a specific schedule. If the initial temperature is high enough to ensure a sufficiently

```
Algorithm SA
begin
    t = t_0;
    cur_part = ini_part;
    cur_score = SCORE(cur_part);
    repeat
        repeat
            comp1 = SELECT(part1);
            comp2 = SELECT(part2);
            trial_part = EXCHANGE(comp1, comp2, cur_part);
            trial_score = SCORE(trial_part);
            δs = trial_score − cur_score;
            if (δs < 0) then
                cur_score = trial_score;
                cur_part = MOVE(comp1, comp2);
            else
                r = RANDOM(0, 1);
                if (r < e^(−δs/t)) then
                    cur_score = trial_score;
                    cur_part = MOVE(comp1, comp2 );
        until (equilibrium at t is reached)
        t = αt     (* 0 < α <1 *)
    until (freezing point is reached)
end.
```

Figure 5.11: Algorithm SA.

random state, and if the cooling is slow enough to ensure that thermal equilibrium is reached at each temperature, then the atoms will arrange themselves in a pattern that closely resembles the global energy minimum of the perfect crystal.

Early work on simulated annealing used Metropolis algorithm [MRR53]. Since then, much work has been done in this field [CH90, GS84, KGV83, RVS84]. Simulated annealing process starts with a random initial partitioning. An altered partitioning is generated by exchanging some elements between partition. The resulting change in score, δs, is calculated. If $\delta s < 0$ (representing lower energy), then the move is accepted. If $\delta s \geq 0$ then the move is accepted with probability $e^{-\frac{\delta s}{t}}$. The probability of accepting an increased score decreases with the increase in temperature *t*. This allows the simulated annealing algorithm to climb out of local optimums in search for a global minimum. This idea is presented as a formal algorithm given by Figure 5.11.

The SELECT function is used to select two random components, one from each partition. These components are considered for exchange between the two

partitions. The EXCHANGE function is used to generate a trial partioning and does not actually move the components. The SCORE function calculates the cost for the new partitioning generated. If the cost is reduced, this move is accepted and the components are actually moved using the MOVE function. The cost evaluated by the SCORE function can be either the cutsize or a combination of cutsize and other factors which need to be optimized. If the cost is greater than the cost for the partitioning before the component was considered for the move, the probability to accept this move is calculated using the RANDOM function. If the move is accepted, the MOVE function is used to actually move the components in between the partitions.

Simulated annealing is an important algorithm in the class of iterative, probabilistic algorithms. The quality of the solution generated by the simulated annealing algorithm depends on the initial value of temperature used and the cooling schedule. Temperature decrement, defined above as αt, is a geometric progression where α is typically 0.95. Performance can be improved by using the temperature decrementfunction, $t = te^{-0.7t}$. However, initial temperature and cooling schedule are parameters that are experimentally determined. The higher the initial temperature and the slower the cooling schedule the better is the result but time required to generate this solution is proportional to the steps in which the temperature is decreased.

5.4.2 Simulated Evolution

Simulated Evolution is in a class of iterative probabilistic methods for combinatorial optimization that exploits an analogy between biological evolution and combinatorial optimization.

In biological processes, species become better as they evolve from one generation to the next generation. The evolution process generally eliminates the "bad" *genes* and maintains the "good" genes of the old generation to produce "better" new generation. This concept has been exploited in iterative improvement techniques for some combinatorial optimization problems [CP86, KB89, SR90, SR89]. In this kind of approach, each feasible solution to the problem is considered as a generation. The bad genes of the solution are identified and eliminated to generate a new feasible solution.

In the following discussion, we present a simulated evolution method, Stochastic Evolution (SE) developed by Saab and Rao [SR90]. SE is introduced as a general-purpose iterative stochastic algorithm that can be used to solve any combinatorial optimization problems whose states fit the certain state model given below.

The state model is defined as follows. Given a finite set M of movable elements and a finite set L of locations, a state is defined as afunction $S : M \rightarrow L$ satisfying certain state-constraints. Also, each state S has an associated cost given by COST(S). The SE algorithm retains the state of lowest cost among those produced by a procedure called PERTURB, thereby generating a new generation. Each time a state is found which has a lower cost than the best state so far, SE decrements the counter by R, thereby increasing the number

of its iterations before termination. The general outline of the SE algorithm is given in Figure 5.12.

PERTURB Procedure: In the biological processes, each gene of a specie in the current generation has to prove its suitability under the existing environmental conditions in order to remain unchanged in the next generation. The PERTURB procedure implements this feature by requiring that each movable element $m \in M$ in the current state S has to prove that its location $S(m)$ is suitable to remain unchanged in the next state of the algorithm. Using the state function model described above, the moves are described as follows. Given S and $m \in M$, a move from S with respect to m is just a change in the value of $S(m)$, i.e., a move generates a new function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$ while $S'(m') = S(m')$ for all $m \neq m' \in M$. A move from a state 5 generates a function $S' : M \rightarrow L$ which may not be a state since it may violate certain state-constraints. This function has to be converted into a state before next iteration begins. The cost function should be suitably extended to include such functions. During each call to PERTURB, the elements of the set M of movable elements are scanned in some ordering. The choice of this ordering is problem-specific.

When element $m \in M$ is being scanned, we assume $S : M \rightarrow L$ be the existing function that may or may not satisfy the state-constraints. A unique sub-move, which is a move from S, is associated with m that generates a new function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$. The details of the sub-move associated with m will be given in below for the partitioning problem. Define $Gain(m) = \text{COST}(S) - \text{COST}(S')$ as the reduction in cost after the sub-move is performed. The procedure PERTURB decides whether or not to accept the sub-move associated with the element m. This decision is made stochastically by using a non-positive control parameter p as follows. The value of $Gain(m)$ is compared to a integer r randomly generated in the interval [p, 0]. If $Gain(m) > r$, then the sub-move to S' is accepted; otherwise, the sub-move is rejected. Since $r \leq 0$, sub-moves with positive gains are always accepted. The algorithm then scans the next element in M. The final function S generated after scanning all elements of M may not satisfy the state-constraints of the problem. In such a case, a function MAKE-STATE(S) is called to reverse the fewest number of latest sub-moves accepted so that all the state-constraints are satisfied. The outline of PERTURB procedure can be outlined as given by Figure 5.12.

Some modifications to the above structure of PERTURB are possible. For example, only a subset M' of M may be scanned in order to save computation time.

The UPDATE procedure: This procedure is used for updating the value of the control parameter p. Initially, p is set to a negative value close to zero so that only moves with small negative gains are performed. It has been observed that moves with large negative gains tend to upset the optimization process and only increase the running time of the algorithm. Hence, the value of p is

```
Algorithm SE
begin
    S = S_0;    (* initial state *)
    S_BEST = S;    (* save initial state *)
    p = p_0;    (* initialize control parameter *)
    γ = 0;    (* initialize counter *)
    repeat
        C_pre = COST(S);
        S = PERTURB(S, p);
        C_cur = COST(S);
        UPDATE(p, C_pre, C_cur);
        if (COST(S) < COST(S_BEST)) then
            S_BEST = S;    (* save best state *)
            γ = γ − R;    (* decrement counter *)
        else
            γ = γ + 1;    (* increment counter *)
    until (γ > R);    (* stopping criterion *)
    return (S_BEST);    (* report best state *)
end.

Procedure PERTURB(S)
begin
    for( each m ∈ M ) do
        S' = SUB-MOVE(S, m);
        Gain(m) = COST(S) - COST(S');
        if Gain(m) > RANDOM(p, 0)
            S = S';
    S = MAKE-STATE(S);
    return S;
end.

Procedure UPDATE(p, C_pre, C_cur)
begin
    if C_pre = C_cur then
        p = p − 1;
    else
        p = p_0;
end.
```

Figure 5.12: Algorithm SE.

reduced only when necessary. During each iteration, the cost C_{cur} of the new state is compared to the cost C_{pre} of the previous state. If both costs are same, p is decremented. Otherwise, p is reset to its initial value. The parameter p is decremented to give the algorithm a chance to escape a local minimum via an uphill climb. The procedure UPDATE is given in Figure 5.12.

Choice of R: The stopping criterion parameter R acts as the expected number of iterations the SE algorithm needs to achieve the objective. The quality of the final state obtained increases with the increase of R. If R is too large, then SE wastes time during the last iterations because it cannot find better states. On the other hand, if R is too small, then SE might not have enough time to improve the initial state.

Let us now discuss the application of SE to partitioning. Using the state model described above, movable elements of a state is the set of vertices, that is $M = V$, and locations of states are the two partitions, that is, $L = 1,2$. A partition therefore is a function $S : V \rightarrow \{1, 2\}$, where the two partitions of the vertex set are the subsets V_1 and V_2. A state (or a bisection) is a partition which satisfy the state-constraint $|V_1| = |V_2|$. Then, the PERTURB procedure scans the vertex set V in some order, i.e., if u and v are two vertices and $u < v$, then u is scanned before v. The sub-move in PERTURB from S that is associated with a vertex $v \in V$ is a move that transfers v from its current partition to the other partition. More precisely, $S' = \text{SUBMOVE}(S, v)$ is an onto function such that $S'(v) = 3 - S(v)$ representing that vertex v is transferred from one partition to another partition and $S'(u) = S(u)$ for all $u \neq v$ representing that all the location of other vertices remain unchanged. Note that S', in general, represents a partition which may or may not be a bisection. After all the vertices have been scanned and the decisions to make the corresponding sub-moves have been made, the resulting function S may not be a state, which means that it may not represent a bisection. Suppose $|V_1| - |V_2| = k > 0$, then MAKE-STATE(S) generates a state from S by reversing the last appropriate $k/2$ sub-moves performed.

The time complexity of SE is proportional to the time required for the computation of the sub-moves and gains associated with each movable element of M. Suppose c is an upper bound on the time required for each sub-move and gain computation, then each iteration of SE runs in $O(c \times |M|)$ time. Since c is either a constant or is linear in the problem size, the SE algorithm for these problems requires either linear or quadratic time per iteration.

The simulated evolution and simulated annealing algorithms are computation intensive. The key difference between these two kinds of algorithms is that the simulated evolution uses the history of previous trial partitionings. Therefore, it is more efficient than simulated annealing. However, it takes more space to store the history of the previous partitioning than the simulated annealing.

5.5 Other Partitioning Algorithms

Besides the group migration and simulated annealing/evolution methods, there are other partitioning methods. In this section, we will present the metric allocation algorithm. The references to other algorithms are provided at the end of the chapter.

5.5.1 Metric Allocation Method

Initial work on measuring the connectivity with a metric was carried out by Charney and Plato [CP68]. They showed that using electrical analog of the network minimizes the distance squared between the components. Partitioning starts after all the values of the metric have been computed; these values are calculated from eigenvalues of the network. This method is described by Cullum, Donath, and Wolfe [CDW75].

The basic metric allocation partitioning algorithm starts with a set V of the nodes and a set $S = S_1, S_2, ..., S_N$ of the nets. A metric value over V x V is computed. Nodes in V are then partitioned into subsets $V_1, V_2, ..., V_k$ such that sum of the areas in V_i is less than or equal to A for all i and the number of nets with members both internal to V_i and external to V_i is less than T for all i, where, A and T represent the area and terminal constraint for each partition, respectively. The algorithm given in Figure 5.13 determines if a k-way partition can be done to satisfy the requirements.

The function CONSTRUCT-ST is used to construct the spanning trees for each net in the netlist. All the edges of these spanning trees are added to a set L by using the function ADD-EDGES. The procedure SORT-ASCENDING sorts L in an ascending order on the metric used. Each vertex v_i is assigned to a individual group G_i by the function INITIALIZE_GROUPS. The groups to which vertices v_i and v_j, joined by edge e_{ij}, belong are collapsed to form a single group if the area and terminal count restriction is not violated. The merging process is carried out by MERGE-GROUPS. This routine also keeps track of the order in which the groups are merged. The function AREA is used to calculate area of a group while function COUNT gives the number of terminals in a group. If such mergings of the groups reduce the number of groups to K or less, the set of groups is returned by the algorithm. If after merging all possible groups, if the number of groups is greater than K, then the smallest group is selected by using function SELECT_SMALL. An attempt is made to merge this group with another group which causes the least increase in area and terminal count of the resulting group. If such a group is found the flag *merge_success* is set to TRUE. The function STORE-MIN is used to store the group which causes the smallest increase in area and terminal count. The function RESTORE-MIN returns the group which is stored by STORE-MIN. If the smallest group consists of only a single component and *merge_success* is FALSE, the algorithm returns a null set indicating failure. If the smallest group consists of more than one component and the *merge_success* flag is set to FALSE, function SELECT_LARGE is used identify the largest group among

```
Algorithm METRIC-PARTITION
begin
    for(i = 1 to N) do
        CONSTRUCT-ST( S_i );
        ADD-EDGES( S_i, L);
    SORT-ASCENDING(L, metric);
    no_groups = INITIALIZE-GROUPS( V );
    while( L ≠ φ ) do
        e_ij = SELECT-EDGE(L);
        if( (G_i ≠ G_j) and
            (AREA(G_i) + AREA(G_j) ≤ A) and
            (COUNT(G_i) + COUNT(G_j) ≤ T) )
                MERGE-GROUPS( G_i, G_j );
                no_groups = no_groups − 1;
                if(no_groups ≤ K)
                    return(G);
        else
            continue;
    while( no_groups > K) do
        G_i = SELECT _ SMALL();
        for(j = 1 to no_groups) do
            if( i ≠ j ) and
                (AREA(G_i) + AREA(G_j) ≤ A) and
                (COUNT(G_i) + COUNT(G_j) ≤ T) )
                    STORE-MIN( G_j );
                    merge_success = TRUE;
        MERGE-GROUPS( G_i, RESTORE-MIN(G_j) );
        if( no_groups ≤ K )
            return( G );
        if( merge_success = FALSE )
            if( SIZE( G_i) = 1 )
                return( φ );
        else
            G_j = SELECT _ LARGE();
            DECOMPOSE( G_j, G_k, G_l );
end.
```

Figure 5.13: Algorithm METRIC-PARTITION.

all groups. This group is decomposed into two subgroups by using function DECOMPOSE and procedure is repeated.

5.6 Performance Driven Partitioning

In recent years, with the advent of the high performance chips, the on-chip delay has been greatly reduced. Typically on-chip delay is in the order of a few nanoseconds while on-board delay is in the order of a few milliseconds. The on-board delay is three orders of magnitude larger than on-chip delay. If a critical path is cut many times by the partition, the delay in the path may be too large to meet the goals of the high performance systems. The design of a high performance system requires partitioning algorithms to reduce the cutsize as well as to minimize the delay in critical paths. The partitioning algorithms, which deal with high performance circuits, are called as *timing (performance) driven* partitioning algorithms and the process of partitioning for such circuits is called timing (performance) driven partitioning.

For timing driven partitioning algorithms, in addition to all the other constraints, timing constraints have to be satisfied. Discussion on these types of partitioning problems for FPGAs can be found in Chapter 11. Timing driven partitioning plays a key role in MCM design and will be discussed in Chapter 12.

The partitioning problem for high performance circuits can be modeled using directed graphs. Let $G = (V, E)$ be a weighted directed graph. Each vertex $v_i \in V$ represents a component (gate) in the circuit and each edge represents a connection between two gates. Each vertex v_i has a weight $GD(v_i)$, specifying the gate delay associated with the gate corresponding to v_i. Each edge (v_i, v_j) has a delay associated with it, which depends on the partitions to which v_i and v_j belong. The edge delay, $ED_{ij} = (d_1, d_2, d_3)$ specifies the delay between v_i and v_j. The delay associated with edge (v_i, v_j) is d_1 if the edge is cut at chip level. If the edge is cut at board level the delay associated with the edge is d_2 and it is d_3 if the edge gets cut at system level. This problem is very general and is still a topic of intensive research.

A timing driven partitioning addresses the problem of clustering a circuit for minimizing its delay, subject to capacity constraints on the clusters. The early work on this problem was done by Stone [Sto66]. When the delay inside a cluster is assumed to be negligible compared to the delay across the clusters, then the following algorithm by Lawler, Levitt and Turner [LLT69], which uses a unit delay model, can be used. The circuit components are represented by a group of vertices or nodes and the nets are represented as directed edges. Each vertex, v_i, has a weight, $W(v_i)$, attached to it indicating the area of the component. A label, $L(v_i)$, is given to each node, v_i, to identify the cluster to which the node belongs. The labeling is done as follows: All the input nodes are labeled 0. A node, v_i, all of whose predecessors have been labeled, is identified. Let k be the largest predecessor label, $WP_i(k)$ be the total weight of all the k-predecessors, and M be the largest weight that can be accommodated in a

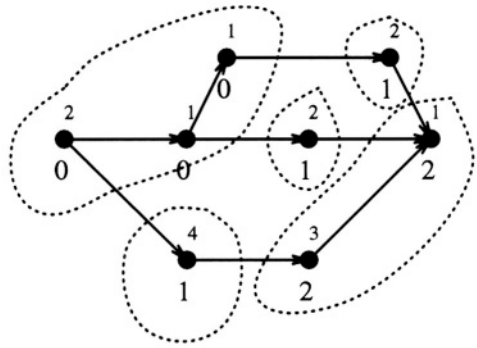


Figure 5.14: Labeling Sequence and clusters formed.

cluster. If $WP_i(k) + W(v_i) \leq M$, label of vertex v_i is set to k, i.e., $L(v_i) = k$, otherwise, vertex v_i gets the label, $k + 1$. After all the vertices are labeled, a vertex, v_j, is identified such that none of the successors of v_j have the same label as v_j. The vertex v_j and all its k-predecessors form a cluster. The vertex v_j is called the root of the cluster. Similar procedure is carried out till all the vertices are clustered. This clustering mechanism may cause a vertex to be in more than one cluster in which case it has to be replicated appropriately. The label for any vertex v_i,as defined above, is the maximum delay of the signal when the signal reaches the vertex v_i after assuming the delay inside a cluster is zero. Thus the above model represents the minimization of the maximum delay of signal under the area constraints when the delay inside a cluster is assumed zero. Figure 5.14 shows a digraph representing a circuit. The number above a vertex indicates the weight of the vertex while the number below a vertex denotes the label of the vertex. M is set equal to 4. Clusters formed are also shown in Figure 5.14.

The clusters (e.g., chips) have large capacities, and very likely, the critical path inside a cluster will be comparable to the total delay of the circuit. Therefore, to be more general, it is better to use more realistic delay model. In a general delay model, each gate of the combinational circuit has a delay associated with it. Considering this problem, Murgai, Brayton and Sangiovanni-Vincentelli [MBV91] proposed an algorithm to reduce this delay. The key idea is to label the vertices (gates) according to the clusters' internal delay. Then number of clusters is minimized without increasing the maximum delay. Minimizing the number of clusters and vertices reduces the number of components and hence the cost of the design. The number of clusters is minimized by merging, subject to a capacity constraint.

5.7 Summary

Partitioning divides a large circuit into a group of smaller subcircuits. This process can be carried out hierarchically until each subcircuit is small enough to be designed efficiently. These subcircuits can be designed independently and simultaneously to speed up the design process. However, the quality of the design may suffer due to partitioning. The partitioning of a circuit has to be done carefully to minimize its ill effects. One of the most common objectives of partitioning is to minimize the cutsize which simplifies the task of routing the nets. The size of the partitions should be balanced. For high performance circuits, the number of times a critical path crosses the partition boundary has to be minimized to reduce delay. Partitioning for high performance circuits is an area of current research, especially so with the advent of high performance chips, and packaging technologies.

Several factors should be considered in the selection of a partitioning algorithm. These factors include difficulty of implementation, performance on common partitioning problems, and time complexity of the algorithm. Group migration method is faster and easier to implement. Metric allocation method is more costly in computing time than group migration method, and hardest to implement since it requires numerical programming. The results show that simulated annealing usually takes much more time than the Kernighan-Lin algorithm does. On a random graph, however, the partitions obtained by simulated annealing are likely to have a smaller cutsize than those produced by the Kernighan-Lin algorithm. Simulated evolution may produce better partition than simulated annealing, but it has larger space complexity. The algorithms for bipartitioning presented in this chapter are practical methods. They are mainly used for bipartitioning, but can be extended to multiway partitioning.

5.8 Exercises

1. Partition the graph shown in Figure 5.15, using Kernighan-Lin algorithm.

†2. Extend Kernighan-Lin algorithm to multiway partitioning of graph.

3. Apply Fiduccia-Mattheyses algorithm for the graph in Figure 5.15 by considering the weights for the vertices, which represent the areas of the modules. Areas associated with the vertices are: $v_1 = 10$, $v_2 = 12$, $v_3 = 8$, $v_4 = 15$, $v_5 = 13$, $v_6 = 20$, $v_7 = 9$, $v_8 = 7$, $v_9 = 14$ and $v_{10} = 9$. The areas of the two partitions should be as equal as possible. Is it possible to apply the Kernighan-Lin algorithm in this problem?

†4. For the graph in Figure 5.16, let the delay for the edges going across the partition be 20 nsec. Each vertex has a delay which is given below. Consider vertex v_1 as the input node and vertex v_8 as the output node. Partition the graph such that the delay between the input node and

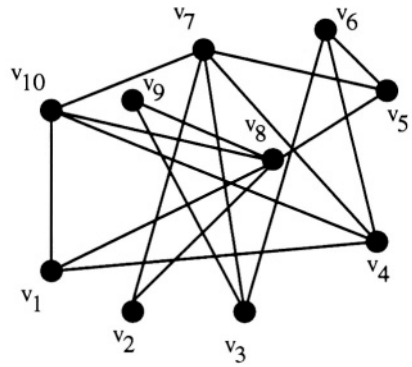


Figure 5.15: A graph partitioning problem.

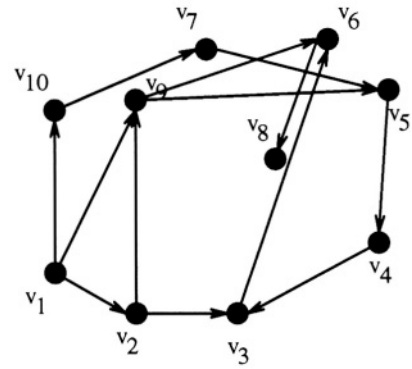


Figure 5.16: A delay minimization problem in graph partitioning.

output node is minimum and the partitions have the same size. The delays for the vertices are $d(v_1) = 3$ nsec, $d(v_2) = 2$ nsec, $d(v_3) = 1$ nsec, $d(v_4) = 2$ nsec, $d(v_5) = 3$ nsec, $d(v_6) = 4$ nsec, $d(v_7) = 3$ nsec, $d(v_8) = 8$ nsec, $d(v_9) = 7$ nsec and $d(v_10) = 5$ nsec.

5. Apply the vertex replication algorithm to the graph given in Figure 5.16.

†6. Implement Fiduccia-Mattheyses and the Kernighan-Lin algorithms for any randomly generated instance, and compare the cutsize.

†7. Implement the Simulated Annealing and Simulated Evolution algorithms described in the text. Compare the efficiency of these two algorithms on a randomly generated example. In what aspects do these algorithms differ?

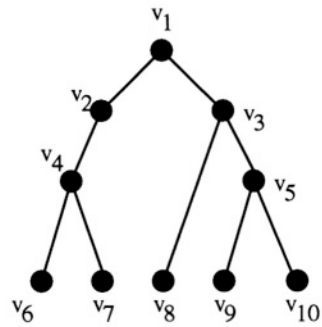


Figure 5.17: A problem instance showing critical net.

8. Compare the performance of the Simulated Annealing algorithm for different values of a.

†9. Consider the tree shown in Figure 5.17, which represents a critical net. Partition the tree into four partitions, two of which will be on one chip and the other two partitions on another chip. Let the delay values of each vertex be the same as that in problem 3. Let the interchip delay be 20 nsec and the delay between the two partitions on the same chip be 10nsec. The objective is to partition the tree into four partitions such that the longest delay path from the root of the tree to any of its leaves is minimized and the number of vertices on each partition is as equal as possible.

‡10. Suggest modifications to the Kernighan-Lin algorithm to speed up the algorithm.

†11. In the application of vertex replication technique, as the vertex replication percentage increases, the cutsize decreases. However, at the same time, the layout area increases as well. Is it possible to get a graph showing the relation between cutsize and circuit area as the component replication percentage varies? Use a randomly generated example. From your result, can you obtain an optimal strategy such that the trade off between cutsize and layout area is compromised?

†12. Implement the ratio cut algorithm. Is it possible to use the vertex replication technique in the ratio cut model?

Bibliographic Notes

Many other partitioning approaches have been proposed in solving circuit partitioning problems, such as network flow [FF62], and eigenvector decomposition [FK86, Hal70] etc. The maximum flow minimum cut algorithm presented by

Ford and Fulkerson [FF62] is an exact algorithm for finding a minimum cost bipartitions for a network without considering the area constraints for the partitions. In many cases, e.g., system level partitioning, the partitioning problem with objective to reduce the number of interconnections, does not represent the actual problem because the area constraints are not considered in this model. Certain algorithms simplify the partitioning problem by restricting the range of the circuits that can be partitioned e.g., partitioning algorithms for planar graphs [Dji82, LT79, Mil84]. Partitioning problem in planar graphs has been discussed in [LT79]. But clearly all circuits cannot be represented as planar graphs. Hence, planar graph algorithms are not very practical in the partitioning of VLSI circuits.

There is an interesting trend in which an interactive man-machine approach is used in solving partitioning problems. Interested readers should read [BKM+66, HMW74, Res86]. A 'Functional Partitioning' which takes into account certain structural qualities of logic circuits, namely loops and reconverging fan-out subnets, can be found in [Gro75]. A new objective function to reduce the number of pins was presented in [Hit70].

A partitioning which intends to form partitions with equal complexity, e.g., similar in terms of area, yield and speed performance, was introduced in [YKR87]. A partitioning model was formulated in which components are assigned probabilities of being placed in bins, separated by partitions. The expected number of nets crossing partitions is a quadratic function of these probabilities. Minimization of this expected value forces condensation of the probabilities into a 'definite' state representing a very good partitioning [Bia89].

A neural network model was proposed in [YM90] for circuit bipartitioning. The massive parallelism of neural nets has been successfully exploited to balance the partitions of a circuit and to reduce the external wiring between the partitions. A constructive partitioning method based on resistive network optimization was developed in [CK84]. Another partitioning technique called clustering was presented in [CB87, Joh67, McF83, McF86, Raj89, RT85]. The simulated annealing algorithm described in this chapter, generates one move at random, evaluates the cost of the move, and then accepts it or rejects it. Greene and Supowit [GS84] proposed an algorithm whereby a list of moves is generated and the moves are taken from the list by a random selection process. In [CLB94] J. Cong, Z. Li, and R. Bagrodia present two algorithms in the acyclic multi-way partitioning approach.