MARINER: A SEA OF GATES LAYOUT SYSTEM

by

Wayne A. Christopher

Memorandum No. UCB/ERL M89/83

28 June 1989

COVER PAGE

MARINER: A SEA OF GATES LAYOUT SYSTEM

by

Wayne A. Christopher

Memorandum No. UCB/ERL M89/83

28 June 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

MARINER: A SEA OF GATES LAYOUT SYSTEM

by

Wayne A. Christopher

Memorandum No. UCB/ERL M89/83

28 June 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

We have written a Sea of Gates layout system which uses an innovative sequence of global and detailed placement and routing steps. The first phase consists of global placement and routing. Global placement partitions the chip area into a grid of "leaf blocks" and assigns the cells to these blocks. This partitioning is accomplished by a modified Kernighan-Lin mincut algorithm known as quadrasection. Global routing by hierarchical refinement is done concurrently with this process. After the partitioning is completed, the global routing is removed and redone using a flat maze-routing algorithm, which yields better results than the hierarchical refinement algorithm. The leaf blocks are then placed and routed individually, using a simulated annealing algorithm that takes into account both netlength and routability, followed by detailed maze routing (currently done by **Mighty**). **Mariner** has been tested on and has yielded good results for small- to medium-sized examples, and our experiments suggest that substantially better results for larger chips could be achieved, given more sophisticated global routing algorithms.

Acknowledgements

I would like to thank Lorraine Layer for her help with the **Mariner** project. She provided a lot of good advice. supplied common sense when I needed it. and most important. was a great friend.

Also. I would like to thank the other Sea of Gates researchers at Berkeley for their assistance. Brian Lee developed the **Hero** router. and provided me with a lot of valuable advice and feedback. Mitch Igusa and Mark Beardslee developed the **Orca** system. which I used to route the output of early versions of **Mariner**. Ren-Song Tsay wrote the **Proud** placement program and provided some of the examples used for this report. I would also like to thank Professor Newton for his support and encouragement. and professor Séquin for valuable advice.

Honorable mention goes to Randy "Xtrek" Cieslak. Abhijit "Tex" Ghosh. Steve "Kitten Heads" Procter. and Leah "Fang" Slyder. all of whom put up with me and are pretty cool people in general.

The members of the CAD group at Berkeley have made the last few years a lot of fun and very educational for me. in particular Professors Brayton, Sangiovanni-Vincentelli. and Pederson. Pranav Ashar. Jeff Burns. Srinivas Devadas. David Harrison. Beorn Johnson. Theo Kelessoglou. Chuck Kring. Tom Laidig. Tony Ma. Sharad Malik. Karti Mayaram. Peter Moore, Brian O'Krafka. Tom Quarles. Rick Rudell. Ellen Sentovich. Kanwar Singh. Rick Spickelmier. and Gregg Whitcomb.

Finally. I would like to thank my wife Susan. my aunt Alice. and Tom. Matt. and Pat for everything they've done for me.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

In recent years, CAD software has been developed for automating the layout of many VLSI design styles. At Berkeley, a number of systems are being developed or are being used for the automated layout of macro-cell [12,4] and standard-cell [59,60] chips. Several systems are currently being developed for the automated layout of Sea of Gates chips, one of which is the **Mariner** system, the subject of this report.

The Sea of Gates design style is similar to gate array, except that there are no pre-defined routing regions. This makes it possible to achieve much higher transistor utilization percentages, especially when more than two customizable layers of high-quality interconnect (e. g. metal, polysilicide) are available.

With this added flexibility comes the requirement for more sophisticated layout algorithms. A number of different approaches have been tried, including a standard cell-like row-based approach, used by the **Proud** system [68], and a macro cell-like approach, used by the **Orca** system [5]. The **Mariner** system uses an innovative combination of top-down partitioning and low-level simulated annealing algorithm for placement, and performs both global and detailed routing concurrently with the placement.

1.1 The Sea of Gates Layout Problem

The Sea of Gates design style may be compared to other ASIC design styles, in particular gate array, standard cell, and macro cell. All these styles use standard *library cells* (called *macros*, for gate array and Sea of Gates), which have been optimized for speed and size. They rely heavily on automated layout systems, with minimal user intervention.

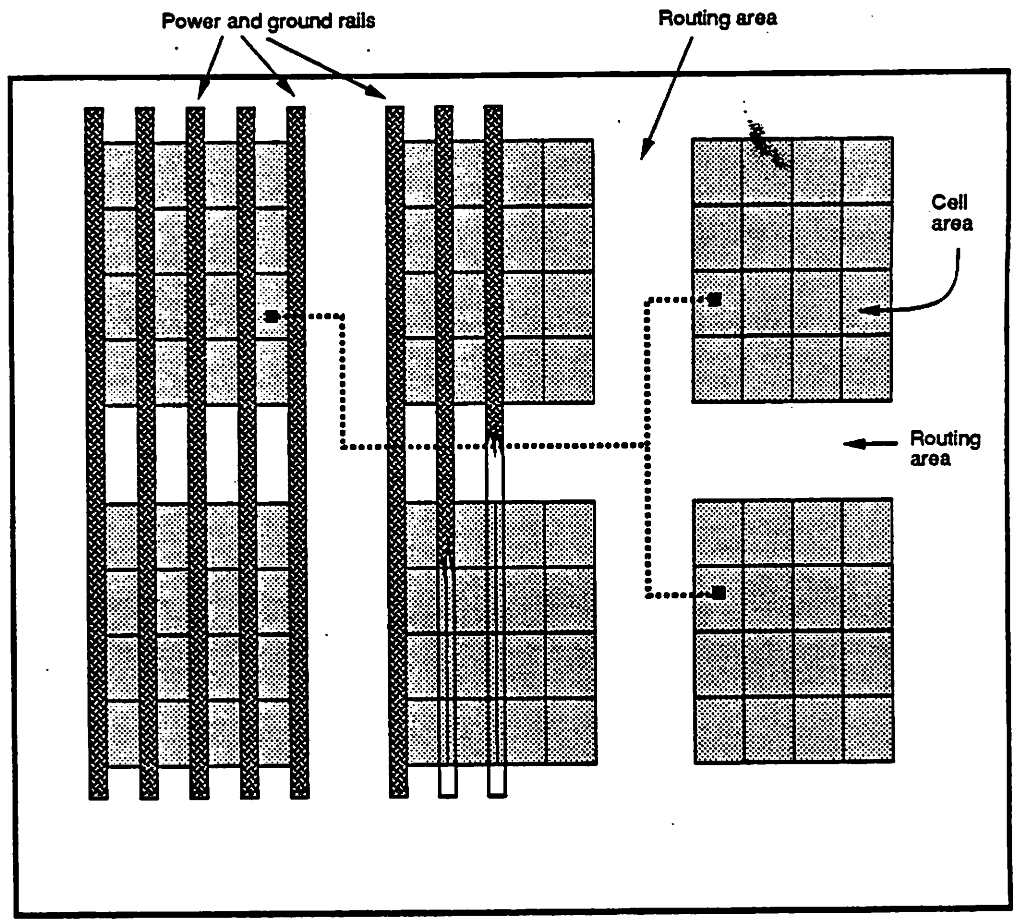

Figure 1.1: The Gate Array Design Style

Finally, they are often used for applications where fast turnaround time is essential, which means that efficient layout algorithms must be used.

The gate array design style, also referred to as master-slice, or uncommitted logic array (see Figure 1.1), is by far the most common programmable array designed by computer. In this approach, a two-dimensional array of replicated transistors is fabricated to a point just prior to the interconnection levels. A particular circuit function is then implemented by customizing the connections within each local group of transistors to define its characteristics as a basic cell, and by customizing the interconnections between cells in the array to define the overall circuit. Generally a two-level interconnection scheme is used for signals and, in some approaches, a third, more coarsely defined layer of interconnections is provided for power and ground connections. The interconnections are implemented on a rectilinear grid in the "channels" between the cells. In many cases, channels are also pro-

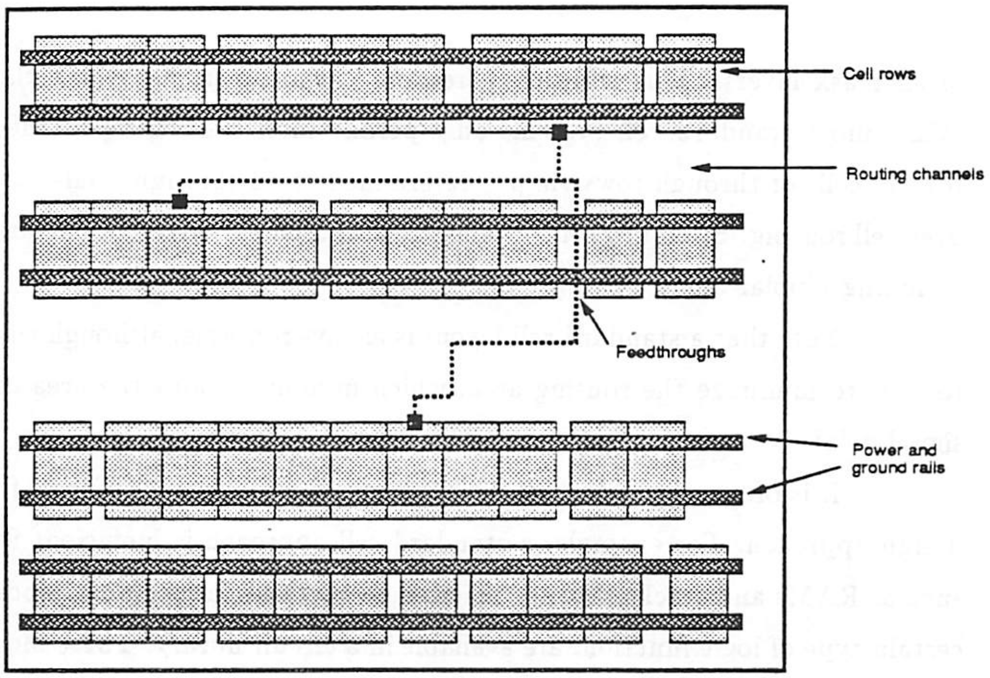

Figure 1.2: The Standard Cell Design Style

vided which run over the cells themselves. and in some arrays, wider channels are provided in the center of the array to alleviate the congestion often found in that area. Gate-arrays are used in many technologies. in particular bipolar and CMOS. and arrays containing many thousands of gates have been fabricated [28.14].

This design style is similar to Sea of Gates. except that gate array defines routing areas. and no cells may be placed in these areas. These additional constraints make it both harder and easier for the layout tools — there is more freedom with Sea of Gates to obtain a better solution. but the placement program must make more decisions. Unlike standard cell. and like Sea of Gates. in a gate array design it is possible to generate a cell placement which cannot be routed. It is also very difficult to modify a gate array placement in any global manner once it has been generated — any change is generally equivalent to a complete re-placement.

The "standard cell" (or polycell) approach (see Figure 1.2) refers to a design method where a library of custom-designed cells is used to implement a logic function. These cells are generally of the complexity of simple logic gates or flip-flops and may be restricted to constant height and/or width to aid packing and ease of power distribution. Unlike the programmable array approach. standard cell layout involves the customization of all mask layers. This additional freedom permits variable width channels to be used. While most standard cell systems only permit inter-cell wiring in the channels between rows of cells or through rows via pre-determined "feed-through" cells. some systems permit over-cell routing. Standard cell systems are also used extensively in a variety of technologies including bipolar and CMOS [51.70].

Note that a standard cell layout is always routable. although the goal of the layout tools is to minimize the routing area. which minimizes both the area of the chip and the signal delay.

It is often relatively inefficient to implement all classes of logic functions in a single design approach. For example. a standard cell approach is inefficient for memory circuits such as RAMs and stacks. In the "macro-cell" method. large circuit blocks. customized to a certain type of logic function. are available in a circuit library. These blocks are of irregular size and shape and may allow functional customization via interconnect. such as a PLA or ROM macro [33]. or they can be parameterized with respect to topology as well [32.8.23]. With the parameterized cell, the number of inputs and outputs may be parameters of the cell. In some systems macro cells may also be embedded in gate-array or standard-cell designs.

The Sea of Gates design style (see Figure 1.3) also makes use of pre-defined polysilicon layers and customizable metal layers. but there are no "bare-silicon" routing areas where cells may not be placed. Rather. the whole chip is covered with pre-fabricated transistors. and it is up to the layout tools to decide where to route nets and where to place cells. In most cases. it is also possible to route nets over the tops of the cells. For example. few cells in the **Mariner** library make extensive internal use of the second layer of metal. and there is at least one vertical track free per transistor where a wire on the first layer of metal may be routed.

The major reason that Sea of Gates is difficult to lay out is that once the cells are put in place. there is no realistic possibility of changing their position — whereas with another design style. the major blocks could be moved slightly apart. with Sea of Gates the minimum amount they can be moved is quite large. This is not a major concern at the detailed level. since it is generally possible to look ahead to the routing stage when doing low-level placement. but at the global level it is a serious problem. If there is a particular area where the congestion is too great. it is not possible to add any tracks without adding a large number of them. across the entire chip. Therefore the rough placement algorithm

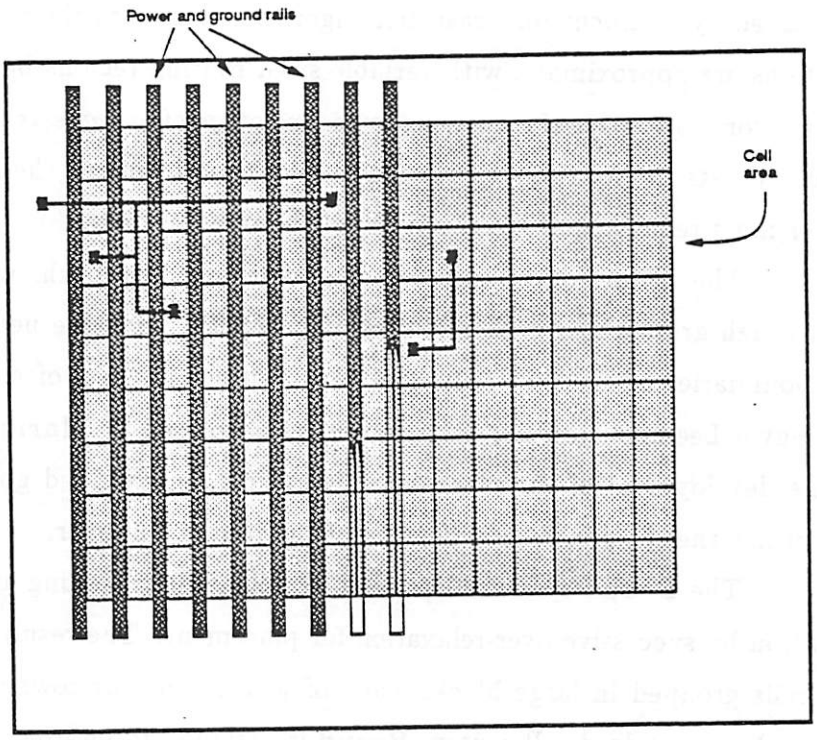

Figure 1.3: The Sea of Gates Design Style

must be very intelligent and look ahead to both the global routing stage and the detailed placement and routing stages.

Cells in a Sea of Gates library cells typically have many feed-throughs and free routing tracks available. Thus much of the routing is over- and through-the-cell. and the detailed router must be able to handle routing problems with arbitrary blockages and pins both on the boundaries of the problem and in arbitrary places within the interior.

Because the transistor structure of a macro is fixed. there is a set of pre-defined positions and transformations on the chip that may be used for a given cell. The placement program must be aware of these constraints in order to generate legal placements. This is not so much a problem for the placement algorithm as an extra level of infrastructure that must be provided for the layout system.

1.2 Existing Sea of Gates Layout Systems

The **Orca** system [5] uses an approach similar to that used by macro cell systems. During the placement, cells are grouped in *clusters* in a hierarchical manner. These clusters are placed by a mincut quadrasection algorithm similar to the one used by **Mariner**. The positions are approximate, with variable-sized routing regions between blocks. The router can ask for more space if necessary, by means of *spacing requests* to the placement program, which adjusts the locations of the macro blocks and returns the layout to the router. The router must request space in multiples of the template size.

The **Orca** router uses a maze-refinement approach, where the nets are routed on a rough grid, which is then successively refined, with the nets generally routed within the boundaries of the grid cells used by the previous level of routing. The basic routing strategy is Lee maze routing. This router was also used by **Mariner** during the early stages of the development of the placement algorithms, and yielded good results. **Admiral** did not utilize the spacing request feature of the router, however.

The **Proud** system [68] uses hierarchical partitioning and resistive network optimization by successive over-relaxation for placement. The result is an unrouted chip with the cells grouped in large blocks, each of which contains rows of cells. The chip is then routed by a standard cell router. **Proud** is designed to be used with three layers of metal [45], and has yielded good results for large industrial designs.

The program **SoGOLaR** [1] (Sea of Gates Optimized Layout and Routing) generates functional cells for static CMOS circuits in the Sea of Gates layout style. The input may be specified either as a list of boolean expressions where each expression specifies a multi-level AOI tree or as a schematic level netlist. For best results it places individual P/N-transistor pairs with simulated annealing using a suitable cost function to minimize netlength and to maximize diffusion sharing. Internal cell routing is performed by **CODAR** [69], a congestion-directed router that combines global and detailed routing algorithms.

A system developed by C. P. Hsu et al. [30] uses techniques similar to those used by **Mariner** and **Orca**. It first flattens the netlist, then separates it into a number of smaller netlists by means of a clustering algorithm. These clusters are then assigned to regions of the chip, and are placed by means of a standard cell simulated annealing placement program, such as TimberWolf [59,60]. One advantage of such a system is that it can easily handle mixed-technology chips. In this case the clustering is based on the functionality of the component, rather than on connectivity alone. Unlike **Mariner**, however, this technique is not fully hierarchical, as it only does one level of clustering. This system has produced good results for large examples (51K cells with 69 % transistor utilization). These results were obtained using Hughes' triple-layer metal process [45], while our examples were produced for the two-layer MOSIS CMOS process [67].

1.3 The Mariner Philosophy

Most prior layout systems have dealt with the place and route problem by first placing the cells. and then routing them. There may be some lookahead during the placement phase to the routing phase. but feedback from routing to placement is usually limited to very coarse changes. such as the expansion of routing channels. This is because placement is usually done in a top-down manner, and high-level decisions cannot easily be modified without invalidating all of the decisions made further down.

Usually both placement and routing in a top-down system are each divided into two phases: global and detailed. During the global phases. more attention is given to large-scale optimization and less is given to low-level considerations such as exact locations and capacities. During the detailed phases, the problems have become much smaller. and it is possible to find exact solutions.

One of the goals of the **Mariner** project was to combine placement and routing into a single process, with much tighter coupling than is usually present in layout systems.

Instead of doing both of the placement phases before the routing phases. both of the global phases are completed before the detailed phases. In the global placement phase. the chip area and the netlist are decomposed into a grid of small placement problems. and during the global routing phase the nets are routed throughout this grid. In the detailed phases, we use simulated annealing to assign positions to the cells. taking into consideration our knowledge of the nets passing through the block. and then route the placed block with a maze router. Note that by doing global routing before detailed placement, we make less information available to the global router. thus making its solution less exact. but we make more routing information to the detailed placer.

In addition to interleaving the placement and routing phases in this manner. both placement phases look ahead to the next routing phase. The global placement algorithm routes nets by means of hierarchical refinement during partitioning, and the detailed placement phase performs pin assignment concurrently with cell placement.

1.4 Overview of Mariner

The **Mariner** system consists of the placement and global routing program. **Admiral**. the cell library. a number of small programs to maintain and update the library cells [41]. and programs to accomplish pad placement and power and ground routing. The detailed router is called by **Admiral**. but it is not considered part of **Mariner**. Currently **Mighty** [62.61] is used for detailed routing. These programs are described briefly below in the order they would be run to generate a completed layout from a netlist view. All these programs read their input from and write their output to *oct* views [24]. with the exception of the human-readable pad placement file.

The program **Mpadp** reads an oct view called *flat*. This view contains cells and pads from the library. none of which is placed. and nets. It also may read a *pad placement file*, in which the relative locations of the pads in the netlist and power and ground pads are specified. along with definitions of power and ground rings and the dimensions of the desired core cell area. If this file is not present. **Mpadp** orders the pads around the chip in the order they were read from the database. connects default power and ground rings. and calculates the area required to achieve a particular transistor utilization percentage. which may be specified on the command line. **Mpadp** outputs an oct view entitled *unplaced*. This view differs from the input view in that the pads now have locations. power and ground pads and rings have been added in the *contents* facet. and a box denoting the available area for cells and routing has been added to the PLACE layer in the *interface* facet of the array.

The **Admiral** program reads the *unplaced* view of the base array and generates either a *placed* view or a *routed* view. depending on whether the user requested routing or not. If routing was not requested. all the cells will have been given legal positions. If routing was requested. in addition. all the signal nets will have been given implementations as well (wires in the METAL-1 and METAL-2 layers connecting all the pins). If **Admiral** could not route some blocks (see Chapter 5), the nets in that block will have been implemented as "spiders" in the PLACE layer. If it could not globally route some nets (see Chapter 3). they will not have any geometry attached to them. and will be attached to a bag called UNROUTED_NETS. Power and ground nets are ignored by **Admiral**.

The power and ground routing program, **Pgroute**, takes a *routed* or *placed* view and adds power and ground busses, connecting them to the power and ground rings around the outside of the chip. It produces a view called *final*. This view is the final output of

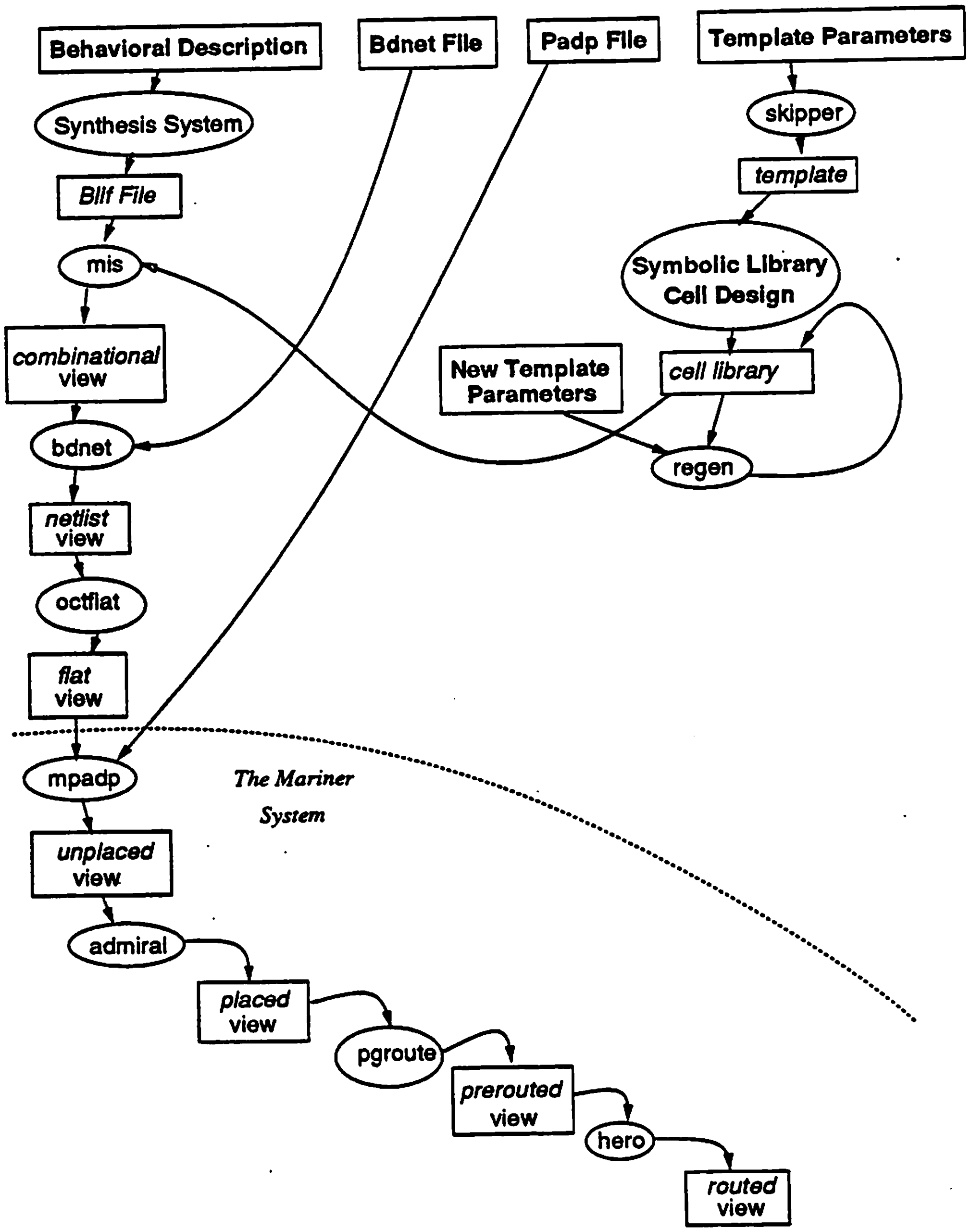

Figure 1.4: The Mariner System

the **Mariner** system. although it can be given as input to another detailed routing program. if routing was not provided by **Admiral**.

A transistor structure (or *template*) and a cell library for Sea of Gates have also been developed and described in detail in [41]. The cell library. based on the Mississippi State University Standard Cell Library [49], contains approximately 30 cells. and is used by the Berkeley logic synthesis tools [7] to generate netlists. The template we developed is shown in Figure 1.5. A 3-input NAND gate from the library is shown in Figure 1.6. There are two programs for maintaining the template and the cell library. These are **Skipper**. which generates a template from chip performance and size specifications and physical design rules. and **Regen**. which modifies the library to fit on a different template than the one it was designed on. Detailed descriptions of these programs may be found in [41].

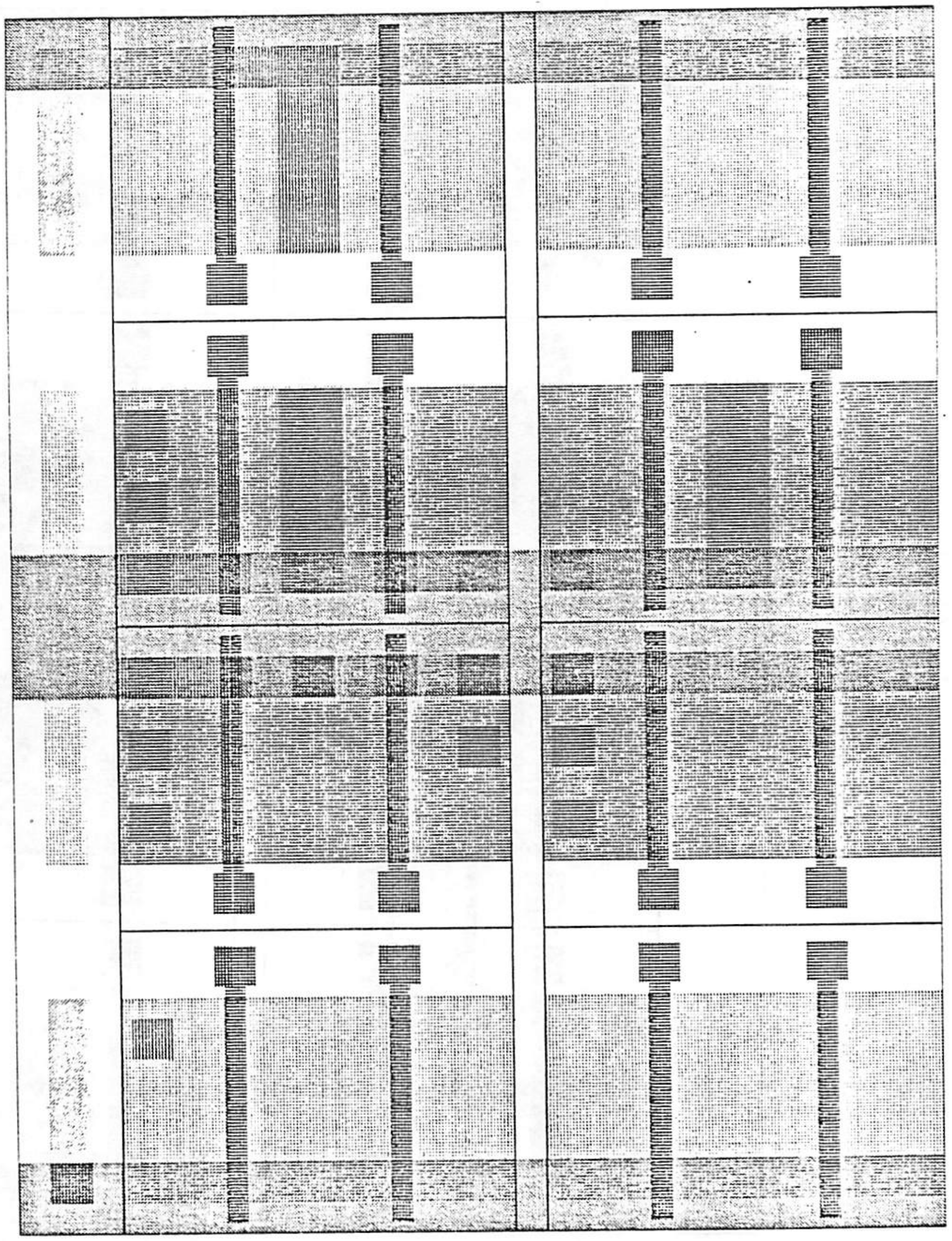

Figure 1.5: Lorraine's Transistor Template

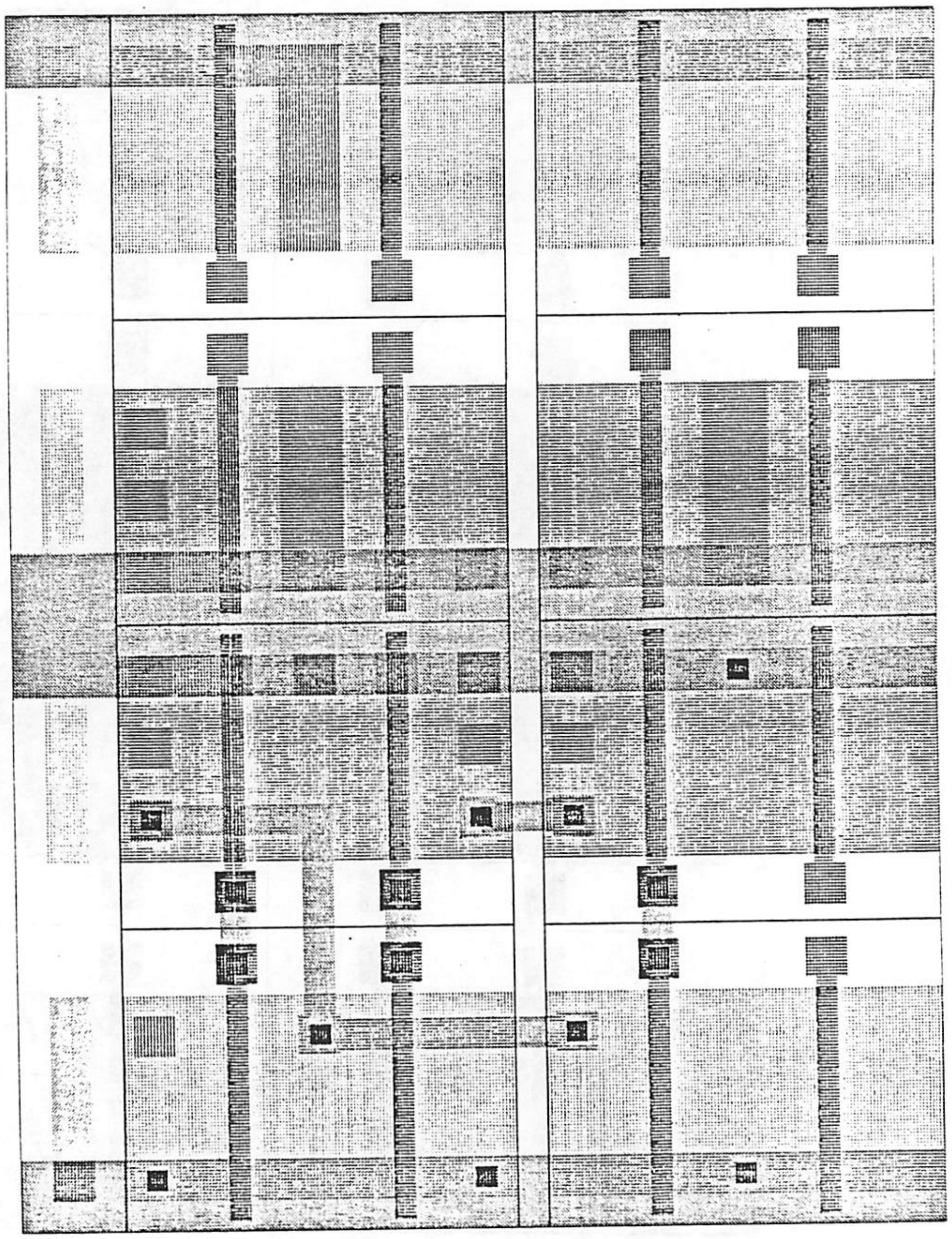

Figure 1.6: Lorraine's NAND3

Chapter 2

PARTITIONING

In order to effectively lay out increasingly large chips, placement algorithms must make use of hierarchy. Some layout systems make use of the hierarchy already present in the design [12.4]. **Admiral** does not use this approach for a number of reasons. First. there may be no such hierarchy. Many of our examples were generated by logic synthesis tools. which are now capable of producing netlists with many thousands of logic gates. Second. the partitioning imposed on the problem by the designer may not be the best partitioning from a layout perspective. It has been suggested [16] that a strict partitioning of a system into datapath and control sections can often lead to a sub-optimal final design. especially in gate array layout styles. Also. although the optimal shape in isolation for a circuit such as an ALU is a regular structure. there is no guarantee that this layout remains optimal in the context of the rest of the chip. especially if parts of the design have been passed through a logic optimizer. For these reasons. we have taken the approach of starting with a flattened netlist and hierarchically partitioning it according to connectivity considerations.

2.1 Partitioning

There are a number of different strategies available for partitioning networks. Among these are *clustering* [13], which is used to identify strongly-connected groups of nodes. and *dissection*. These algorithms are not suitable to partitioning a netlist for placement purposes, since we usually want to split the netlist into a small number of pieces of approximately equal size. and clustering and dissection gives us little control over the number or size of the partitions. A hybrid approach. using some clustering followed by partitioning. has been successful for placement [55]. and might also prove useful for Sea of Gates layout [11].

Our basic algorithm is a variant of min-cut partitioning known as *quadrasection* which was first used by Suaris and Kedem for standard-cell layout [65]. Quadrasection is used to split the netlist and the chip area into four pieces, and is then applied recursively to the sub-netlists until a set of "small" problems is obtained. What is meant by "small" is discussed in Section 2.4.

We investigated a number of algorithms for partitioning a netlist. subject to cell capacity constraints and external pin positions. The algorithms are described below. and the results are summarized in Table 2.3. Also. we tried an alternate formulation of the problem. where the quantity being minimized was wiring congestion. rather than total edge crossings. The results of this experiment are also described in Section 2.3.

2.1.1 The Problem

The N-way mincut partitioning problem can be described as follows. The inputs are a list of nets, a list of cells, and a list of partitions. A cell may reside in exactly one partition, and is considered a point to which the nets are attached. The information that may be specified about these items is as follows:

- nets
  1. list of cells connected to the net
  2. the *criticality* of the net. an externally specified value between 0 and 1000. which is used as a weighting factor in various places
- cells
  1. cell area. in square wiring grid units
  2. initial partition cell resides in
  3. is cell fixed in that partition?
- partitions
  1. total area available for cells
  2. distance to all other partitions, using some metric

The problem is to place the cells in partitions. subject to the maximum cell area capacities of the partitions. in such a way as to minimize the total cost of the nets. The

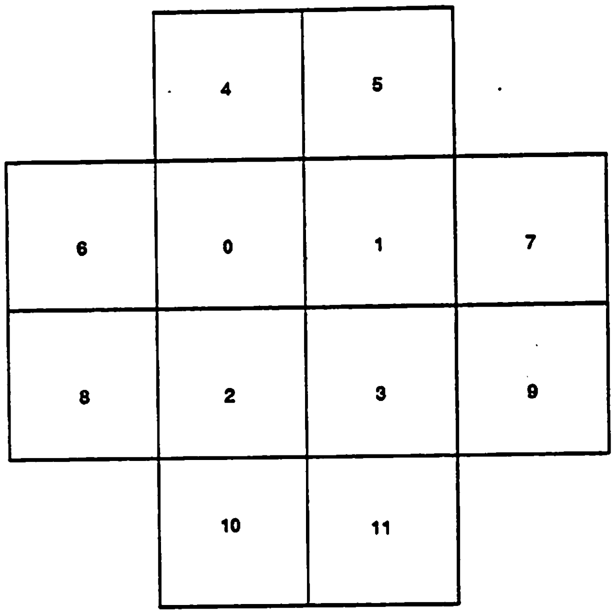

Figure 2.1: Partitions

cost of a net may be measured in a number of ways: if there are only two partitions, then if the net contains cells in both partitions, it costs one unit, otherwise it costs nothing. The cost may be scaled by the net's criticality. If there are more than two partitions, the cost becomes more complex to determine, since the net really should be routed to see which partition boundaries it falls upon. Since doing an actual global route each time a cell is moved would be prohibitively slow, this process is simulated by using the minimum spanning tree of the partitions occupied. This is calculated using Kruskal's algorithm [2], and the results are cached for efficiency.

The Placement Problem

We must map the placement problem at a particular level to this abstraction of the partitioning problem. In order to do this some extra partitions must be constructed. The placement algorithm produces a problem consisting of four "real" partitions and eight "fake" partitions, which are introduced to contain external pins. The arrangement of these partitions are shown in Figure 2.1. Partitions 0 through 3 are empty, and represent the actual cell area available. Partitions 4 through 11 each contain one dummy cell, which is fixed in place. All nets that have external pins that reside on a particular side contain the dummy cell in the corresponding partition. If a pin is free to float along the top, for instance. the net connects the cells in partitions 4 and 5. and if it is constrained to the right side of the top edge (because the adjacent block has already been partitioned. or the pin corresponds to a pad which has a fixed position), it contains the cell in partition 5. Note that if information about the partitioning of a block is to be used to influence the partitions of neighboring blocks. the net containing that pin must be routed in the 2×2 problem produced by the partition. For this reason, after each block is partitioned. the nets contained in it are globally routed by means of a hierarchical refinement algorithm. described in Section 3.1.

We can justify our use of Kruskal's algorithm to calculate the minimum *spanning* tree of the complete graph. instead of the minimum *Steiner* tree of the graph induced by the adjacencies of the partitions. by the way the placement code calculates the distances between partitions. The distance between a "fake" partition and a "real" partition that are adjacent is defined to be 1. The ratio of the distances between pairs of real partitions that have horizontal boundaries to those between pairs that have vertical boundaries is calculated to be roughly the same as the ratio between the vertical and the horizontal net and cell blockage factors. because we want to capture the relative difficulty of routing horizontal and vertical nets in the distance values. For all other pairs of partitions. the distance is defined to be the total distance along the minimum path between the partitions. using edges that already have distances defined. By doing this the cost of the minimum Steiner tree connecting the partitions is "pre-calculated".

Note that this formulation ignores the problem of congestion. It assumes that the minimum cost path for a net will always be available, while in practice not all nets that wish to take this path may be allowed to. The *minimum density* partitioning algorithm (see Section 2.3) was an attempt to address this facet of the problem more directly.

Unused Area

Another important issue is that of free area, which will be utilized for routing. If the partitioning algorithm is given the actual value of the amount of room available in each of the partitions, there will be no control over where the unused area appears. The goal of mincut partitioning is to minimize the number of edge crossings. and this may be done at the expense of distributing unused area very unevenly. For this reason. the partitioning algorithm must be given partitions whose total size is not very much larger than the total area of the cells to be placed. If a total amount of extra room equal to the number of real partitions times the maximum cell size is left in the partitions, we can be sure that all the partitioning algorithms described below will always be able to find a possible move. The problem then becomes where to add the remaining area, or *slack*.

Slack should be added to partitions based on the perceived difficulty of routing the sub-problem that will be produced. The amount of slack that each partition gets should be proportional to its *difficulty value*. Estimating this value is more difficult than estimating routing area (Section 3.2), since we don't even know what cells or nets are present in this partition. There are a few ways this value can be estimated, however. The first is based on the distance from the center of the block to the center of the chip. Since more nets tend to be routed through the center of the chip than the edges, a partition closer to the center will have a higher difficulty value and will need more routing area. This scheme has the advantage of being simple. On the other hand, the number of nets going through a partition can easily be measured more directly. All that is known at this point is the number of pins on each edge, and in some cases to which half of the edge the pin is restricted (if a neighboring block has already been partitioned). Define the *difficulty* of a partition p to be the sum for all the pins t on all the edges ϵ of that partition of $f(t)$, where

$$f(pin) = \begin{cases} 0.5 & \text{if } t \text{ is floating} \\ 1 & \text{if } t \text{ is fixed to the half of } \epsilon \text{ adjacent to } p \\ 0 & \text{if } t \text{ is fixed to the half of } \epsilon \text{ not adjacent to } p \end{cases}$$

That is, if it is known that a net must pass through this partition, it is given full weight, and if it is known that it might do so, it is given half weight. Thus the partitions that are likely to have a lot of external pins will get more slack.

This heuristic can be improved a bit more, however. A net entering a partition may or may not leave it. If a net enters a partition only to connect to a pin on a cell in that partition, it will take less routing area than a net that must exit on the other side. For this reason, a bit more can be added to $f(t)$ if it is known that the net the pin is on has other external pins. A value of 0.25 has proved to work well.

In Table 2.1, these three methods of calculating difficulty are compared. The numbers given are the maximum and average net densities after the second phase of global routing is completed. The third method, taking into account the number of nets that both enter and leave the partition, seems to be the best overall. For these results, and in general

Circuit Name	Center Distance	Nets Entering	Nets Entering and Leaving
adder32	0.6 / 0.4	0.5 / 0.3	0.6 / 0.4
pr1	1.0 / 0.4	0.9 / 0.5	0.8 / 0.6
square	0.8 / 0.3	1.0 / 0.3	0.8 / 0.3
des	1.1 / 0.5	1.3 / 0.5	0.9 / 0.5

Table 2.1: Comparison of Difficulty Functions

for net densities. the maximum values are more important than the average values. since a maximum value over 1.0 means that the routing cannot be done.

Another method of dealing with slack is to begin with all partitions having the same amount of slack, and then iterate, at each stage using the predicted wiring space required by the previous partition to determine how much slack to give it for the current iteration. The process would terminate when the quality of the solution. as measured by the "fairness" of the slack distribution, ceases to improve. We did not test this technique. but the iterative control structure for the partitioning described below has a similar effect.

2.2 Quadrasection

The quadrasection algorithms tested were random assignment. clustering. generalized Kernighan-Lin-Fiduccia-Matheyses (KLFM) partitioning. Suaris-Kedem quadrasection. simple simulated annealing. adaptive simulated annealing. and adaptive simulated annealing using net moves instead of cell moves. Note that all these algorithms. with the exception of Suaris-Kedem quadrasection, can partition a netlist into an arbitrary number of partitions, but our tests have been done for 4 partitions in all cases.

2.2.1 Random Partitioning

Random assignment is simple: each movable cell is assigned in succession to a random partition that has space available. In general, care must be taken that a configuration isn't created that has enough total room for the last block. but no single partition has enough. However, the placement algorithm is careful to adjust the sizes so that this cannot happen.

2.2.2 Clustering

Clustering is performed as follows. A seed cell is first picked for each of the partitions at random. Then for each partition in succession the free cell which is most strongly connected to the other cells is added to that partition. if there is enough available space. This process is repeated until there are no more free cells. In order to determine the most highly connected cell for each partition. gain tables are maintained. much like those used in the KLFM algorithm.

2.2.3 Kernighan-Lin-Fiduccia-Matheyses

The Kernighan-Lin-Fiduccia-Matheyses (KLFM) algorithm [34] is conceptually simple. although its efficient implementation can be rather tricky. For the two-way case. the algorithm is as follows:

```
WHILE (improvement made in cut-set size) {
    WHILE (cells remain to be moved) {
        c = the cell whose move to the other
            partition would lead to the greatest
            decrease in the size of the cut-set;
        move c;
    }
    restore the configuration to the best found this pass:
}
```

The step of picking the cell with the best gain can be performed in constant time, using a bucket-sorting strategy which was described in [20]. Briefly. for each partition. an array of lists of *cell moves* from that partition to the other is maintained. where the position in the array corresponds to the *gain*, or decrease in cut-set size, that would be obtained if a move in that list were made. Since the range of possible gains is proportional to the maximum number of pins on a cell. the size of the array is relatively small. (If net criticalities are taken into account. the arrays would become larger.) Fiduccia and Matheyses have shown a way to update these gain tables in constant time whenever a move is made, which leads to the overall linear runtime of the KLFM algorithm.

The generalization to N partitions is straightforward [65]. The cost the sum of the sizes of the spanning trees, as described above. instead of the size of the cut-set. Instead of finding the best cell to move, the algorithm must pick the best (*cell. partition*) pair.

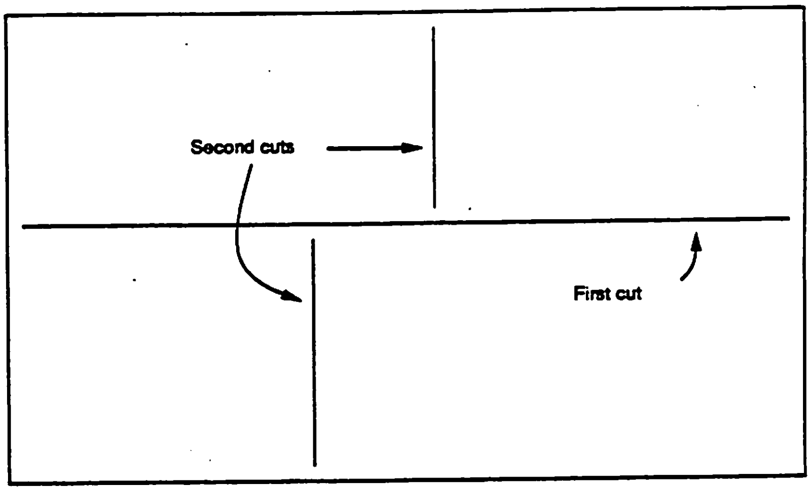

Figure 2.2: Quadrasection

where *partition* is the destination of the cell. For N partitions, a total of $N \times (N - 1)$ gain tables is required. The minimum cost move is chosen from these which do not violate the capacity constraints of the destination partition. If no move can be made which does not violate capacity, one move that does violate capacity is allowed, since the violation will very likely be eliminated by a future move. Note that although this situation cannot arise in the original partitioning problem produced by the placement algorithm, it can occur in one of the sub-problems produced by the quadrasection algorithm.

This algorithm is an *improvement* algorithm, and it needs a good starting partition in order to work well. We tried both the random and cluster algorithms for initial partitions, and both results are given in Table 2.3.

2.2.4 Suaris-Kedem Quadrasection

The quadrasection algorithm is essentially a control strategy for KLFM. It is specific to the case where the number of real partitions is four, although the technique could be easily generalized. The area to be partitioned is first divided into upper and lower halves, and then each of those halves is partitioned into right and left halves. (See Figure 2.2.) Previous placement algorithms have taken this as the final partition [18], but following [65], we apply a further 4-way improvement step. Statistics for the partition with and without this improvement step are given, as are statistics for random assignment and clustering as the initial strategy for the 2-way partitions. These statistics show that the improvement

Parameter	Value
initial temperature	100
moves per temperature per cell	10
temperature decrease per point	0.9
points without an accept before stop	3
last temp point cap-violating moves allowed	90

Table 2.2: Simulated Annealing Partitioning Parameters

step usually leads to an improvement in the final solution of between 10% and 30%. and using the clustering algorithm usually yields better solutions. although not as dramatically better as the improvement step.

2.2.5 Simulated Annealing

Simulated annealing is used both here and in the detailed placement phase. The general simulated annealing algorithm is described in Section 4.2. This section describes the application of simulated annealing to quadrasection.

The first simulated annealing algorithm makes use of fixed parameters for the cooling schedule and the inner-loop stopping criterion. This technique was used by early placement algorithms [35.59]. and although the adaptive algorithm described below yields better results, the fixed algorithm is much simpler. The parameters used are given in Table 2.2.

One move consists of moving a cell from one partition to another. Cells that cannot move are not considered for movement. and "fake" partitions are not considered as destinations. The cost function is the spanning tree cost described above. Moves are allowed to violate capacity, with a penalty of 1000 times the capacity violation. After a certain temperature point, such moves are disallowed. After a certain number of temperature points without a move accepted, the process is terminated.

This is a relatively simple annealing algorithm. and the parameters given above were chosen empirically, based on a small set of examples.

2.2.6 Adaptive Simulated Annealing

The general adaptive simulated annealing algorithm is described in Section 4.2. It uses the same move set and cost function as the regular simulated annealing algorithm described above. and will not be discussed further.

2.2.7 Adaptive Simulated Annealing with Net Moves

The second adaptive simulated annealing algorithm uses *net moves* instead of cell moves. Instead of choosing a cell to move from one partition to another. this algorithm selects a net and a destination partition. and then moves all the movable cells on that net into that partition. The rationale behind this is that the goal is to maximize the number of nets wholly contained in one partition. moving a net at a time seems like a more direct way to achieve this goal. Simulated annealing was chosen as the control framework for this algorithm since it was the easiest to use — an iterative hill-climbing algorithm like Kernighan-Lin might have worked better, but if the concept of net moves is a good one, it stands to reason that it would yield good results in the adaptive simulated annealing framework also.

2.2.8 Results

The results obtained by partitioning some large netlists with these algorithms are given in Table 2.3. The corresponding runtimes are given in Table 2.4. These times were obtained on a VAX 8650.

Note that although some of the results for the simulated annealing algorithms are comparable to the results of KLFM and quadrasection. the runtimes become prohibitively large for even medium-sized examples: no problems with size over 1000 cells were able to complete in a reasonable amount of time. Simulated annealing has been empirically found to take time that is roughly quadratic in the problem size to achieve good results. while KLFM runs in linear time.

Among the annealing algorithms, the adaptive annealing algorithm did the best. probably because of the finer control over the cooling schedule. The net-move algorithm also did worse than the regular adaptive algorithm. Although this could be in part a result of the adaptive algorithm's unsuitability as a control framework for this move set. it is probably an indication that the moves in the move set are too "coarse-grained". in the

Circuit Name	Num Cells	Random	Cluster	KLFM Rand	KLFM Cluster	Quad Rand	Quad Cluster	Simple Anneal	Adapt Anneal	Net Anneal
pr1	841	1697	554	406	323	253	254	—	—	—
pr2	3022	3022	2190	1590	1107	1085	1050	—	—	—
hughes	19724	35582	9843	9426	4496	4235	1861	—	—	—
square	2508	8370	1543	983	1065	854	854	—	—	—
adder8	70	172	142	96	98	92	94	—	—	—
adder16	191	538	523	283	289	254	245	—	—	—
adder32	384	1143	1110	524	503	441	465	—	—	—
mult32	5979	10484	4555	2764	1575	1811	1117	—	—	—
rand100	100	1740	1091	689	649	682	616	732	682	760
rand200	200	3885	2274	1612	1481	1343	1338	1522	1396	1507
rand500	500	9315	5477	3933	3894	3573	3666	3912	3608	3836
rand1000	1000	18614	11072	7439	7228	6961	6901	7187	7177	7798
rand5000	5000	93085	54694	38890	39033	36704	36529	—	—	—

Table 2.3: Partitioning Algorithm Results: Costs of Final Solutions

Circuit Name	Num Cells	Random	Cluster	KLFM Rand	KLFM Cluster	Quad Rand	Quad Cluster	Simple Anneal	Adapt Anneal	Net Anneal
pr1	841	1.3	1.5	9.0	7.6	12.0	14.1	—	—	—
pr2	3022	5.1	6.1	50.0	29.2	101	56.7	—	—	—
hughes	19724	43.2	46.6	658	274	604	320	—	—	—
square	2508	4.8	5.0	29.1	26.0	35.2	32.7	—	—	—
adder8	70	0.1	0.1	0.3	0.3	0.2	0.4	—	—	—
adder16	191	0.3	0.3	1.4	1.4	2.2	1.8	—	—	—
adder32	384	0.6	0.7	5.0	5.7	3.3	4.6	—	—	—
mult32	5979	9.7	12.0	76.0	43.5	136	89.3	—	—	—
rand100	100	0.1	0.2	0.7	0.4	0.6	0.6	44.3	21.0	47.7
rand200	200	0.2	0.2	1.3	1.0	1.8	1.2	103	88.0	208
rand500	500	0.7	1.1	3.2	3.4	5.6	3.9	374	765	1396
rand1000	1000	1.5	1.5	9.2	9.3	9.5	9.9	8179	3724	2966
rand5000	5000	6.9	7.9	51.8	39.1	98.0	51.8	—	—	—

Table 2.4: Partitioning Algorithm Times. in Seconds

sense that they change the configuration too drastically. There are not enough small moves to move out of "shallow bowls" [64] without moving too far, when the temperature is low. Perhaps an approach that provided both the large-scale net moves and the smaller-scale cell moves would work better.

It has been noted [34] that the KLFM algorithm is very sensitive to the initial partition given. This can be seen by comparing the columns "KLFM Cluster" with "KLFM Random", and "Quad Cluster" with "Quad Random". The clustering algorithm is much better than random partitioning, and for the larger examples, this difference in starting quality leads to a difference in quality of final results. For the smaller examples there seems to be little if any difference in the final quality. Also, in most of the examples the "KLFM" and "Quad" algorithms that use clustering for the initial partition take less time than the corresponding algorithms that use random initial partitions. This is because a better original solution leads to fewer passes.

The example "square" is an artificially constructed netlist, which could ideally be laid out in a rectangular 50×50 mesh, each cell connecting only to its four neighbors. Thus the ideal partition should have 50 vertical and 50 horizontal crossings on the center lines. Although this cannot be seen from the data in Table 2.3, due to the weighting factors introduced by the placement algorithm and extra crossings due to external connections, the 4-way KLFM algorithm produced 56 and 54 horizontal and vertical crossings, respectively, and the quadrasection algorithm produced 52 and 52 crossings. A result so close to the known minimum for even one example gives confidence in the essential soundness of the algorithm.

2.2.9 Control Strategy

We tested a number of different control strategies for applying the partitioning algorithm. The first was depth-first: after a block is partitioned, partition each of its children in turn. This turned out to be too order-dependent, since the first sub-tree partitioned had much more freedom than the later sub-trees. The second approach tried was ordering all the sub-problems on a particular level according to their estimated complexity. This also turned out to be too order-dependent. The approach we settled upon is similar to that used by Suaris and Kedem, which is to partition the blocks, and then iterate ripping up and re-partitioning the blocks until there is no more improvement in quality. At that point

Circuit Name	Depth First max / avg	Random max / avg	Est. Complexity max / avg	Iterated Rip-up max / avg
adder32	0.88 / 0.55	1.11 / 0.58	0.81 / 0.53	0.79 / 0.53
pr1	1.02 / 0.51	1.30 / 0.57	0.72 / 0.49	0.71 / 0.49
pr2	1.10 / 0.67	1.25 / 0.65	0.89 / 0.52	0.85 / 0.51
mult32	1.05 / 0.70	1.70 / 0.70	0.91 / 0.61	0.93 / 0.62
rand1000	2.21 / 0.98	3.42 / 1.03	1.41 / 0.75	1.32 / 0.72
rand5000	3.04 / 1.20	4.24 / 2.29	2.04 / 0.97	1.75 / 0.78

Table 2.5: Ripup and Re-partition Results

the next level down is partitioned. This approach was combined with initially ordering the sub-problems according to estimated complexity. The results of this comparison are given in Table 2.5. The figures are for the maximum and average global routing densities on the chip after the final global route is completed. The methods for estimating complexity are discussed in Section 3.2. The iterated rip-up algorithm performed the best overall. although ordering the blocks by estimated complexity did almost as well in many cases.

2.3 Minimum Density Partitioning

Another approach to partitioning that we tried was minimum-density partitioning. Preliminary trials indicated that this algorithm was not very promising. and we did not have time to implement it efficiently enough to collect extensive statistics on it. The algorithm is described below.

We made the observation that mincut partitioning does not take into account the complexity of the wiring within the partitions. nor does it consider the wiring density at the boundaries between partitions. There are cases where a partition has a lower total cost. using the min-cut metric. yet is less routable (has a higher maximum density) than one with a higher cost (see Figure 2.3). This lead us to ask whether we could devise a good partitioning algorithm that uses density directly as a cost function instead of cutsize. and if so. whether it would yield better results than mincut. in which the cost function is more loosely correlated to the value we want to optimize.

The cost function we chose has two terms in it. These were the number of tracks left in each partition. $T_{h,i}$ and $T_{v,i}$. for $i \in \{1.2.3.4\}$. and the cell area violations in each

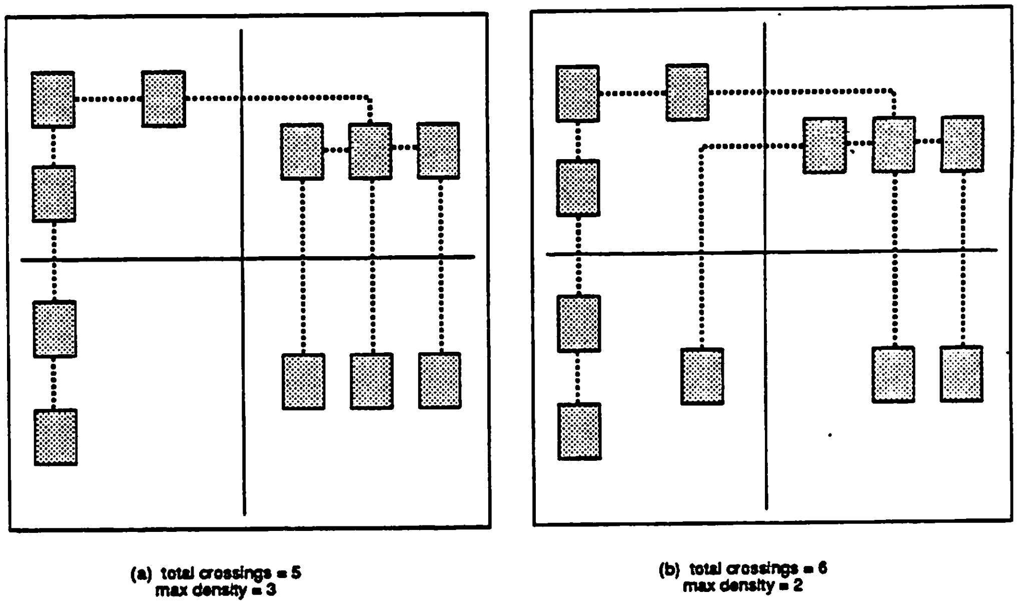

Figure 2.3: Minimum Cut Vs. Minimum Density Partitions

partition. C_i. The cost is defined by

$$\text{cost} = W_{cell} \sum_{i=0}^{3} C_i + W_{wire} \sum_{i=0}^{3} (\text{adjust}(T_{h,i}, H_{max}) + \text{adjust}(T_{v,i}, V_{max}))$$

where W_{cell} and W_{wire} are 10 and 1 respectively. The function *adjust* is designed to give more weight to partitions that have close to 0 free tracks, and is shown in Figure 2.4. Note that this cost function assumes that the nets have already been globally routed, so the global routing information must be updated each time a cell move is made.

The terms in the cost function are straightforward to compute. The techniques for determining the number of tracks free are described in Section 3.2. The difficult part is computing the change to these values efficiently when a cell move is made. This can be accomplished by keeping careful track for each cell and net of what contributions they have made to each of the relevant values.

The available moves are cell moves and possibly cell swaps and net moves. Since none of the other partitioning algorithms uses cell swaps, and net moves were found to be ineffective, the moveset was limited to simple cell moves. After a cell is moved, the nets attached to that cell are globally rerouted in the 2 x 2 grid.

Another major consideration is the speed of such an algorithm. KLFM partitioning has O(n) runtime per pass. For another partitioning algorithm to be practically useful, its

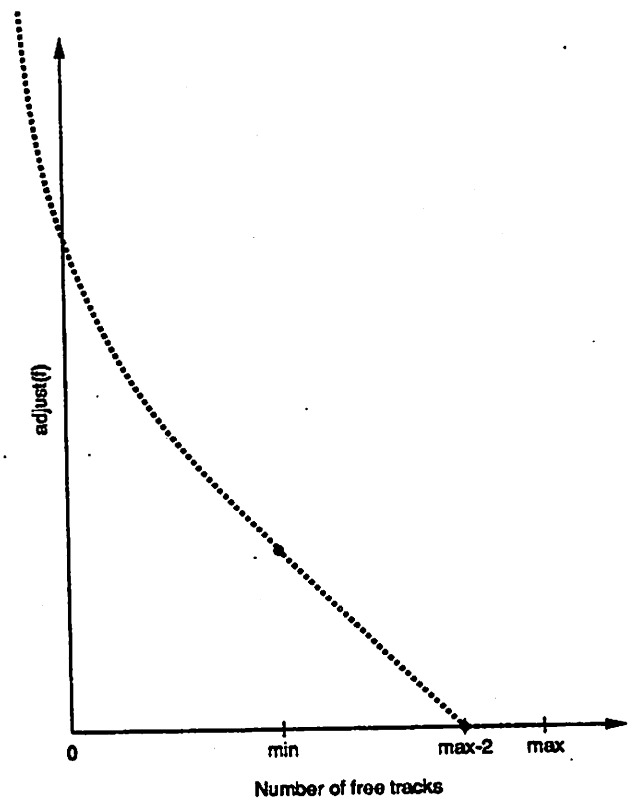

Figure 2.4: The Function *adjust*

Circuit Name	No. Cells	Min Dens Cost	Min Dens Runtime	Min Cut Cost	Min Cut Runtime
adder8	62	53	7	50	1
adder16	191	449	121	317	8
adder32	384	972	673	837	18
pr1	841	3341	1991	2593	37

Table 2.6: Results of Minimum Density Partitioning

efficiency must be close to this. However. for testing our minimum density partitioning algorithm we used the adaptive simulated annealing framework. Although we would have to devise a better control strategy eventually. this experiment gave some indication of what sort of results we could expect.

This implementation was tested on some small examples. We compared both the runtime and the final quality with the adaptive annealing mincut partitioning algorithm. which has the same control framework. The results, which are given in Table 2.6. were not encouraging. They might be interpreted as follows. The density. although a more accurate cost function, is not only more difficult to maintain but is too uniform. The difference in density between two configurations will in general be much less than the difference in crossing count. since the density takes into account the cell and net blockages. which change more slowly than the crossings during the course of partitioning. Although minimum density partitioning eliminates the need for some of the heuristics that try to predict where the congestion will be, such as the division of "slack" before partitioning is done. this is not necessarily a good thing. It seems that in this situation. as is often the case. it is better to solve some sub-problems that approximate the real problem very well than to find an inferior solution to the real problem.

2.4 When to Stop Partitioning

Partitioning process terminates when a block that is a good candidate for detailed placement and routing is reached. There are a number of factors that must be considered when deciding whether a block is a leaf problem or not.

The first and most important one is whether partitioning the block further would make it impossible to place the cells in the block. Since all the cells in a block must reside

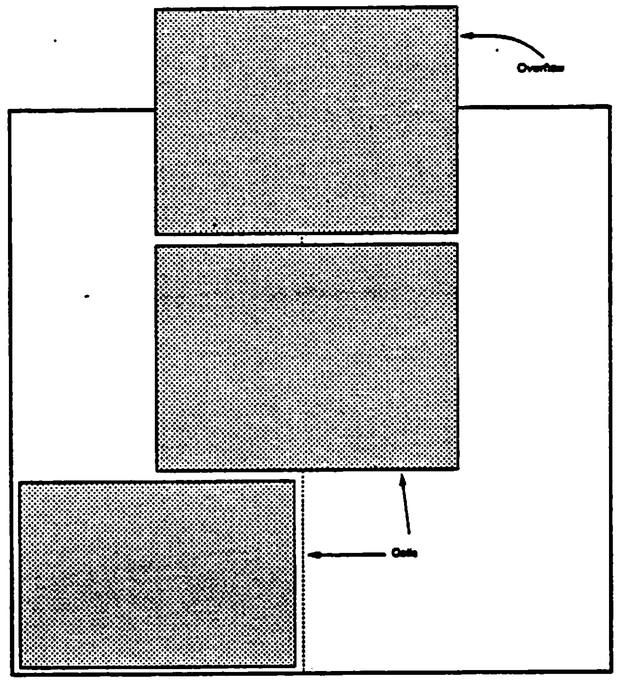

Figure 2.5: A Block That is Too Small for its Cells

entirely within the block. if the block is smaller than one of the cells. or small enough that the cells cannot be arranged within the block. even though the total area of the cells is less than the block area (as in Figure 2.5). its parent should not have been partitioned.

Also. partitioning a block restricts the possible layouts. Cells cannot stretch across block boundaries. and a net can have only one pin per edge of a leaf block (although after the second phase of global routing. it may have more than one pin on an edge of a higher-level block). To reduce the effects of bad partitioning decisions. we tried using a "rip-up and re-place" heuristic. This consists of collapsing one level of the partition hierarchy and trying to solve the resulting larger problem.

This proved ineffective for several reasons. A placement problem that is four times larger takes much more than four times longer to run. and **Mighty** likewise scales more than linearly. Also, if a block is very difficult to lay out. the chances are that its neighbors will also be very difficult. so the overall complexity will not decrease much. Finally, because of the way we do the second phase of maze routing. when we collapse one level of block. nets might have more than one pin on an edge of the leaf block. This situation cannot be represented by the data structure currently used by the detailed placement system.

Based on these considerations. we have adopted a simple scheme for ending the partitioning process. A block is considered a leaf block if either its width or its height is less than or equal to 4 times the width or height respectively of one *basic tiling unit* (see Figure 1.5), or of the largest cell in the block, whichever is larger. This makes it very unlikely that the sort of situation pictured in Figure 2.5 can occur.

Chapter 3

GLOBAL ROUTING

There are two phases of global routing performed before the detailed placement and detailed routing phases. These are initial hierarchical routing and global maze routing. The initial routing is performed purely to direct the partitioning process. The final global routing phase is performed after the initial routing has been ripped up. and uses a variation of the Lee maze-routing algorithm.

3.1 The First Global Routing Phase

The initial global routing uses a hierarchical approach. similar to those of Burstein [10] and Suaris and Kedem [65]. It is performed concurrently with the partitioning process. After each block is partitioned. all the nets which pass through this block are assigned routes through the 2×2 grid created by the quadrasection. Note that although we call this a 2×2 routing problem, we also need to determine which direction a net takes from one of the partitions to the outside of the grid. Thus this is more general than the 2×2 problem as described in [42]. A net may have pins on cells in any of the four sub-blocks. and may have a pin on any of the four sides. Furthermore. the pin may be floating along an edge. or restructed to either the first or the second half. (See Figure 3.1.) The pin location restriction may be a result of either an adjacent block which is already routed, in which case it is called *half-floating*, or an external pin on the edge of the chip by the position of the corresponding pad. in which case it is called *fixed*.

These conditions may be expressed as an 8-tuple $\{P_0. P_1. P_2. P_3. E_t. E_b. E_l. E_r\}$. where the P_i's are either 1 or 0 depending on whether the net has a pin in the i'th partition.

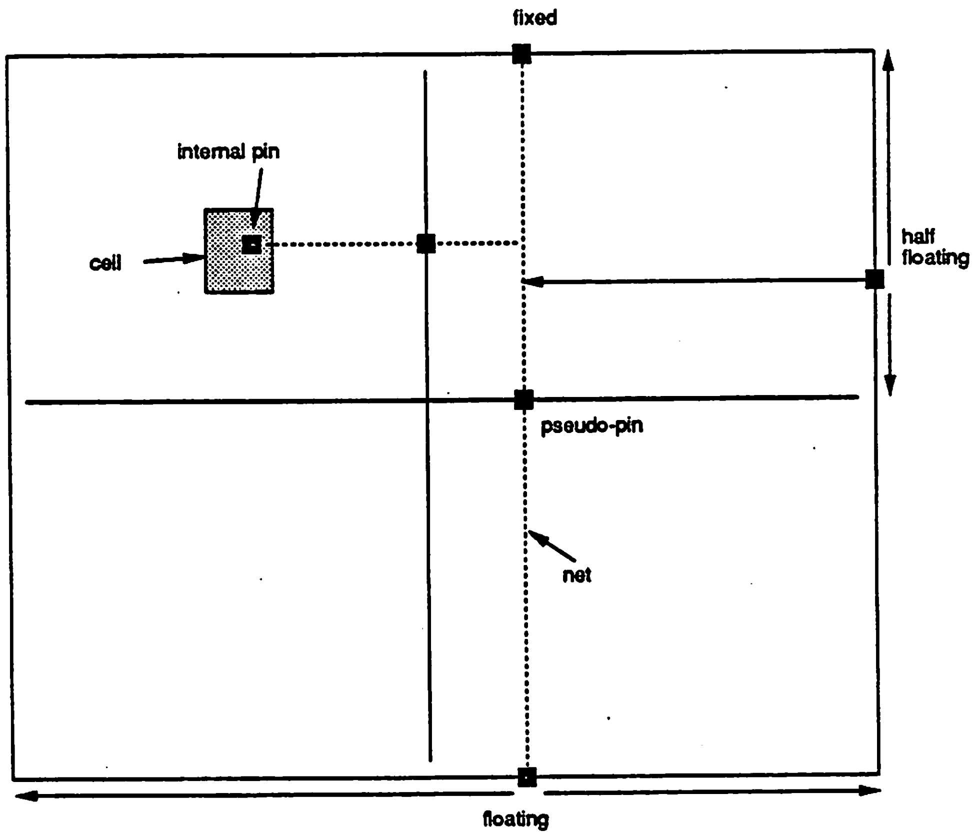

Figure 3.1: The 2 × 2 Routing Problem

and the E_i's take on one of the values {**none, upper, lower, floating**}. Thus there are 2^{12} possible net types to route. Since enumerating all the possible routes for a particular net type may be a time consuming operation. a "memo function" is used to calculate the routes. which remembers all its past return values in an array of size 2^{12}. Enumerating the possible routes for a particular configuration is not excessively expensive. though — the maximum number of routes is 256. but reasonable 2×2 routing problems have only a few solutions.

The possible routes are calculated in the following manner. First. each of the 12 edge halves (internal and external) are marked with one of {**yes, no, either, maybe**}. where **yes** indicates that there must be a pin on that edge half. **no** indicates that a pin may not be on that edge half. **either** indicates that a pin must be on either that edge half or the other half of the edge. and **maybe** indicates that there may or may not be a pin on that edge half. The choices now are first. for every edge half that is **either**. whether to put the pin there or the other half. and second. for every pin that is **maybe**. whether to put a pin there or not. All such choices are then enumerated. and those that either contain cycles, contain disjoint components, or don't contain all the partitions that contain cells on the net are rejected.

A possible route for a net may be expressed as a 12-tuple $\{X_0.X_1.\ldots X_{11}\}$. where X_i is 1 iff there is a pin on edge number i. If i is an internal edge. the pin is a *pseudo-pin.*

These choices are then given costs, and the lowest cost route for the net is chosen. We investigated two types of cost functions. The first takes into account the congestion at each edge. and sets the cost equal to

$$\sum_{i \in edges} \pi(i) f(ratio_i)$$

where $ratio_i = density/capacity$ at edge i,

$$\pi(i) = \begin{cases} 1 & \text{if the net has a pin on edge} i \\ 0 & \text{otherwise} \end{cases}$$

and

$$f(r) = \begin{cases} r & \text{if } r < C_{max} \\ F_{soft} r & \text{if } r >= C_{max} \text{ and } r < 1 \\ F_{hard} r & \text{if } r >= 1 \end{cases}$$

C_{max}, the desired maximum capacity, is a value determined empirically to be 0.75 by examining the success patterns of the detailed placement and routing. F_{soft} (the "soft"

Cost Function	F_{soft} Value	F_{hard} Value	Max / Avg Density	Max / Avg Net Size
Weighted	5	20	1.00 / 0.57	156 / 6.48
Weighted	10	100	1.04 / 0.54	161 / 6.48
Weighted	10	20	1.09 / 0.55	164 / 6.53
Weighted	3	20	1.10 / 0.55	160 / 6.36
Weighted	20	100	1.14 / 0.54	155 / 6.53
Weighted	3	1	1.15 / 0.55	160 / 6.36
Weighted	2	1	1.21 / 0.56	152 / 6.45
Weighted	2	100	1.21 / 0.56	152 / 6.46
Weighted	1	100	1.33 / 0.55	155 / 6.42
Weighted	1	3	1.47 / 0.55	155 / 6.42
Uniform	-	-	0.92 / 0.54	157 / 6.41

Table 3.1: Comparison of Initial Global Routing Cost Functions

penalty factor) and F_{hard} (the "hard" penalty factor) are 5 and 20 respectively, and are likewise empirically determined.

The second cost function uses the same sum, with $f(ratio) = 1$. That is, the route with the smallest number of crossings is always chosen.

Results for these two cost functions are given in Table 3.1, for some values of F_{soft} and F_{hard}. The second one works better although it takes no account of the congestion. The reason for this is that the purpose of this global routing phase is to direct the partitioning process. If many nets want to cross an edge on one half instead of the other, they should be allowed to cross there so that the next round of partitioning can take this fact into account, instead of being forced to take detours, which hides the better routes from the partitioning. The fact that the routes may violate capacity constraints does not matter, since these global routes are not actually used. Furthermore, using the cost function $f(ratio) = 1$ eliminates the problem of order dependence in this phase of global routing, since the value of the function doesn't depend upon the previous density of the edges crossed by the nets.

3.2 Routing Area Prediction

After the partitioning process is finished, we wish to estimate the routing capacities of each of the leaf blocks as accurately as possible, before doing the final global route of the nets. The X- or Y-capacity of a block is defined as the total number of routing tracks in that

Number of Pins	Equivalent Nets
2	1
3	1.2
4	1.4
5	1.6
6	1.8
7	2.0
8	2.0

Table 3.2: Empirically Determined Multiple Pin Equivalent Nets

direction (not counting those used by power and ground busses). minus the total estimated blockages. An estimated blockage may be either a blockage from a cell or a blockage from a net internal to the block. Determining this value accurately is very difficult. since the placement is not known. so we err on the conservative side whenever necessary.

Blockages due to cells within the block may be estimated in a straightforward fashion. The total length of the obstacles in the appropriate direction is calculated. and then this is divided by the width or height of the block to produce the best-case number of tracks that would be blocked. That is. if there are a total of $K \mu$m of vertical wires in cells of the block. and the height of the block is $H \mu$m. K/H tracks are blocked. If a wire runs in the wrong direction. it is modelled as a series of length 1 wires.

This number is then multiplied by K_{cell}. an empirical constant which measures how much the actual value deviates from the best-case value calculated above. This has been determined to be approximately 1.4. The actual number of blockages is measured by noting how many external wires can be routed through this block by the detailed router.

Blockages due to nets are calculated in a more interesting manner. First. all multi-pin nets are expressed as a number of two-pin nets. which are equivalent with respect to netlength. The empirically derived values for these numbers are given in Table 3.2. Note that external pins are excluded from this count, since the number of available external connections to nets is precisely that which we are trying to determine.

The total number of equivalent two-pin nets. N_{equiv}. is then multiplied by the average netlength for this block. which is calculated by the following formula. due to Donath et al[17,25], who have derived it from Rent's rule[39].

$$H_{avg} = K_{net}C^{p-0.5}\sqrt{(C)}\frac{H_{free}}{(H_{free} + V_{free})}$$

$$V_{avg} = K_{net}C^{p-0.5}\sqrt{(C)}\frac{V_{free}}{(H_{free} + V_{free})}$$

where

C	= number of cells in the block
H_{free}	= height – horizontal tracks blocked by cell blockages
V_{free}	= width – vertical tracks blocked by cell blockages
K_{net}	= empirical constant, found to be approximately 1.2
p	= "parallelism factor" for netlist

The parallelism factor for the netlist is usually between 0.5 and 0.75 for typical netlists. We do not attempt to estimate the parallelism factor, but rather allow the user to specify a value for it. After the layout is completed, the actual value found for this constant is printed, and the user may supply this to successive placement runs.

The total estimated routing tracks available for the block are thus

$$\text{Vertical tracks} = width - pgtracks - V_{cell} - V_{net}$$

$$\text{Horizontal tracks} = height - H_{cell} - H_{net}$$

where

H_{cell}	= the total number of horizontal cell blockages
V_{cell}	= the total number of vertical cell blockages
H_{net}	= the total number of horizontal net blockages
V_{net}	= the total number of vertical net blockages
$pgtracks$	= number of tracks used for power and ground
$height$	= height of the block in tracks
$width$	= width of the block in tracks

We have found this technique to be reasonably good for large sub-problems, but it becomes less accurate when it is applied to small sub-problems where the effects of detailed routing are much more pronounced. However, this effect is minimized by the fact that the global routing algorithms tend to work better on a small scale than a large scale, and the "rip up and re-place" technique can be applied. The above formulas were derived for standard-cell designs — a careful examination of their validity for gate array types of designs might lead to a better better interconnection model for Sea of Gates.

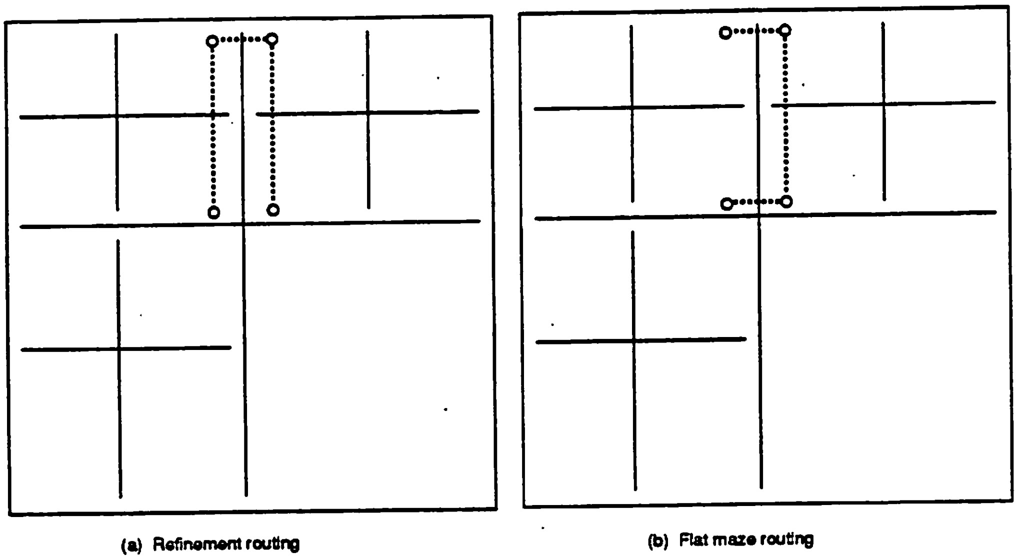

Figure 3.2: Hierarchical vs Flat Routing

3.3 The Second Global Routing Phase

After the partitioning process is completed, all the nets that span more than one block are ripped up and globally rerouted. This is done because the initial, four-way refinement global routing algorithm must make decisions early in the partitioning process that drastically affect available routes later on, without enough information to make the best choices. Also, the types of routes that can be generated by the simple refinement algorithm used here are a subset of the reasonable routes that one might consider. In Figure 3.2, route (a) is the best route that can be generated by our hierarchical refinement, but route (b) is a much better route. The reason that our hierarchical routing algorithm cannot generate route (b) is that at each level of the hierarchy, a net can cross any edge only once. Note that there are techniques for hierarchical routing that relax this restriction [44], but these are more complex and time-consuming. Instead of improving our hierarchical routing algorithm, we decided to use it only for directing the partitioning, and re-route all the nets after the partitioning has been completed.

3.3.1 The Problem

The routing problem is as follows. The area to be used for routing is a grid, as in Figure 3.3a, each of which has a certain number of vertical and horizontal feedthroughs.

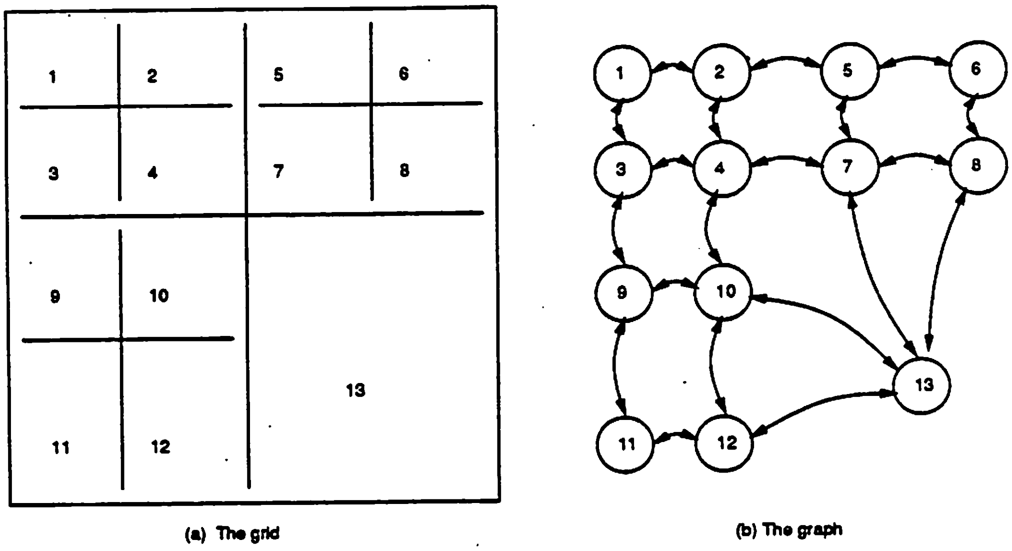

Figure 3.3: Two Representations of the Global Routing Problem

These numbers are calculated by means of the routing area prediction algorithm described in Section 3.2. For any boundary between adjacent blocks. the capacity of that boundary is defined as the minimum of the number of horizontal or vertical feedthroughs for the two blocks. for vertical and horizontal boundaries, respectively. This grid can be thought of as an undirected graph. with edges having capacities (Figure 3.3b).

Nets contain pins which reside in graph nodes. External pins. which correspond to pads. are contained by the node corresponding to the block nearest to the pad. A net can contain pins in an arbitrary subset of nodes. The global routing problem thus becomes that of assigning Steiner trees to each net. such that some function of the sizes of the individual Steiner trees and the *density/capacity* ratios of the arcs in the graph is minimized.

This problem is in general an NP-complete problem. There has been a great deal of research to develop heuristic algorithms to solve this problem. none of which have been entirely satisfactory [38].

3.3.2 Maze Routing

One of the better and simpler classes of algorithms for solving this problem in its graph formulation is the Lee maze-routing algorithm [43.50]. The Lee algorithm and the similar Hightower line-search algorithm [26] have also been successfully applied to detailed

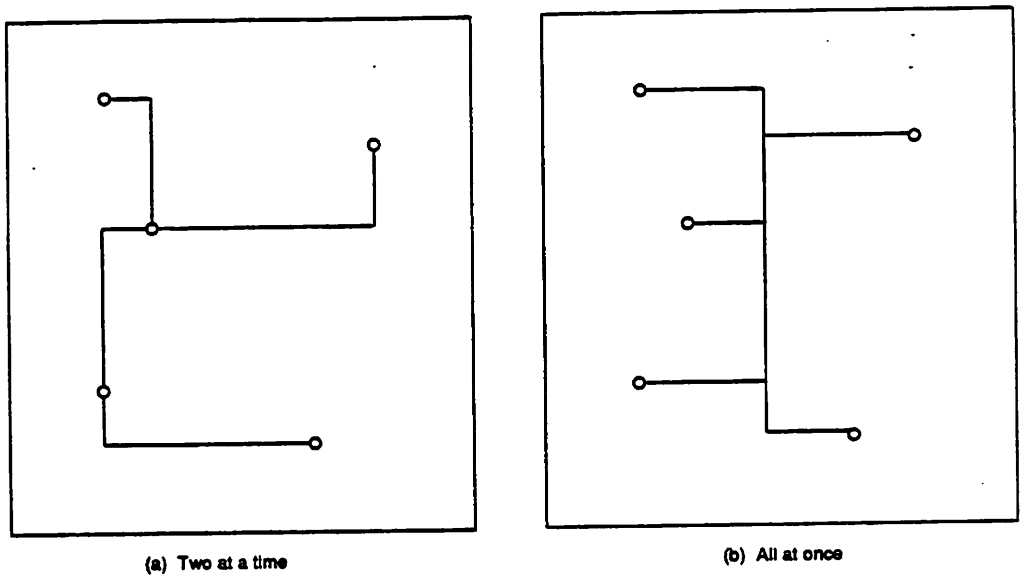

Figure 3.4: Different Ways To Route A Multi-Pin Net

routing [62]. The essential idea of the Lee algorithm is to expand from one or more nodes in the graph. keeping track of the path to each node so that once the destination is reached the path taken can be reconstructed.

In the two-terminal case. one can pick one terminal and expand towards the other. possibly biasing the search in directions that are more likely than others to yield minimal routes. In the more general case. where a net may have any number of pins. there are several choices. The problem may be decomposed into a set of two-terminal problems. where each segment of the net connects two pins (Figure 3.4a). However. splitting the net in this manner introduces another problem that must be solved — how are the pairs of pins to be connected chosen. and in what order should they be connected?

The solution we have chosen is to route the multi-terminal net all at once. as shown in Figure 3.4b. Connection points between wires may also occur in places other than the terminal locations. which changes the problem from one of finding a *spanning* tree into one of finding a *Steiner* tree. Finding a minimal size Steiner tree has been shown to be NP-complete [22]. so a heuristic algorithm similar to that used to solve the two-terminal problem is applied.

The routing problem is represented by a graph $\mathcal{G}$. a set of *terminal nodes* $\mathcal{T}$ in the graph. and a collection $\mathcal{V}$ of subsets of the graph. each of which is connected. Each of the subsets $\mathcal{V}_i$ contains at least one element of $\mathcal{T}$. and represents the nodes reached from that element. Thus at the start of the algorithm. each $\mathcal{V}_i$ consists of a single element of $\mathcal{T}$. Additionally. a set of *junction nodes* $\mathcal{J}$ is maintained, where the $\mathcal{V}_i$ meet during the expansion process. initially ϕ. a function $\mathcal{L}$ which indicates for each node (except the terminal nodes) in a member of $\mathcal{V}$ which neighbor it was reached from. and a function $\mathcal{K}$ of the nodes which indicates which element of $\mathcal{V}$ that node is in. if any. Finally. there is a priority queue $\mathcal{F}$ of the nodes on the *frontier* of the maze search.

The objective of the algorithm is to expand all of the $\mathcal{V}_i$ and whenever two subsets meet. combine the two. adding the node at which they met to $\mathcal{J}$. When there is only one subset remaining. retrace the paths from each node in $\mathcal{J}$ to the terminal node. and add this path to the route for the net. The algorithm is as follows.

```
$\mathcal{J} = \mathcal{F} = \mathcal{V} = \mathcal{L} = \mathcal{K} = \mathcal{R} = \phi$;
FORALL ($t \in \mathcal{T}$){
    $\mathcal{V} = \mathcal{V} \cup \{\{t\}\}$:
    insert $t$ into $\mathcal{F}$;
}
WHILE ($|\mathcal{V}| > 1$){
    $n$ = first($\mathcal{F}$):
    FOREACH (neighbor $m$ of $n$) {
        IF ($\exists i$ s.t. $\{m.i\} \in \mathcal{K}$){
            merge $\mathcal{K}(m)$ and $\mathcal{K}(n)$;
            $\mathcal{J} = \mathcal{J} \cup \{m\}$;
        } ELSE {
            $np = \mathcal{K}(n)$;
            $\mathcal{V}_{np} = \mathcal{V}_{np} \cup \{m\}$;
            $\mathcal{K}(m) = \mathcal{K}(n)$;
            add $m$ to $\mathcal{F}$;
        }
    }
}
FORALL ($j \in \mathcal{J}$){
    $\mathcal{R} = \mathcal{R} \cup \{j\}$;
    WHILE ($\exists x$ s.t. $\{j.x\} \in \mathcal{L}$) {
        $\mathcal{R} = \mathcal{R} \cup \{x\}$;
        $j = x$;
    }
}.
```

The major decision at this point is what cost function to use for the priority queue. In the case of a constant cost function, the priority queue becomes a regular queue. and the algorithm becomes the classic Lee algorithm that finds the shortest topographical path. This can be very slow. however. and fails to take into account the costs of edges. To speed up the algorithm. the search process can be biased towards routes that are likely to be optimal. and against ones that are likely to be expensive (such as those that lead in the opposite direction from the goal). In the two-terminal case. we can start at one node. and add nodes according to their distance from the other node. This is the idea behind A^* routing [56]. If the routing is done from both ends at the same time. this heuristic can still be applied. If. however. there are more than two terminals. it is not so easy to apply. since there is no one location that is the goal. It is much easier to take the opposite approach. namely considering how far the frontier is from the starting point. as opposed to how far it is from the ending point.

The edges in the graph have weights. which depend on the density of the corresponding edges in the global wiring grid. A weight of ∞ indicates that the edge is blocked. and a weight of 0 indicates that the edge is "free". Since the total cost of a route is defined as the sum of the weights of the edges that comprise the route. this quantity can be minimized by making the cost function for the priority queue the sum of the weights of the edges used to reach the node. Therefore. every time a node is removed from the queue. the cheapest path available is being expanded.

However. the cheapest path may not always be the most desirable. In the control strategy described below. at the beginning it is more important to find the best possible path for each net. independent of the others, which would correspond to a constant cost function. Near the end. the goal is to find the lowest cost path. In order to achieve this. an extra term was added to the cost function. which is the cost of entering a node. This cost is the same for all nodes in a particular pass of the maze routing. We tried both keeping this value constant. and starting with a high cost for entering a node. then decreasing it to zero in the following passes. Finally, yet another term was added to the cost function. which is proportional to the total cost of the arcs leaving the node. The rationale here is that it should be tough to enter a node that will be difficult to leave. Here also we tried using a constant factor and one that decreases in successive passes. The values of these functions at the different passes are shown in Table 3.3 . The results of comparing the constant and the varying values, as well as tests to determine the optimal constant factors. are described below.

Finally. when the neighbors of this node are enumerated, it is done from the least

Pass Number	Enter Cost	Arc Factor
1	100	0.5
2	60	0.3
3	30	0.1
4	10	0.0
5	0	0.0

Table 3.3: Cost Weights for Entering Nodes

expensive to the most expensive. so that if there is a choice of junction points. the best one will be picked.

3.3.3 Control Strategy

The above maze routing algorithm routes one net. This leaves the problems of what order to route the nets in. and what to do if no path can be found. This is an extensively studied problem. and no generally satisfactory solution exists yet. We tried a number of strategies, which are described below. Results are given in Table 3.5. Several elements can be factored out and varied independently — initial routing order. ripup selection strategy. termination criteria. and the calculation of edge weights from the *density/capacity* ratios at the boundaries.

Initial Ordering

The simplest strategy to implement is an arbitrary ordering. or routing nets in the order they appear in the database.

The second strategy is to route the shortest nets first. The rationale behind this heuristic is that a short net will block fewer edges than a long one, and if the long nets are routed first too many short nets may be blocked. Also. a long net will have more reasonable alternative routes available.

The third strategy is to route the longest nets first. The rationale behind this heuristic is that there will be fewer long nets than short nets, and they will be harder to route since there are more edges that can be congested. so it is better to route them before routing the shorter. simpler nets.

Note that none of these strategies consider net criticality. If a net is marked as critical. it should be routed earlier than nets which are not so critical. and should get a more optimal route. but not at the expense of making other nets unroutable.

Ripup Selection Strategy

If some of the nets have no path available. it is necessary to rip up the nets that block it and reroute them. Some researchers [48] have expressed the belief that if an algorithm is forced to do any large amount of ripping up. there is no hope of completing the routing. However. in the absence of an initial routing strategy that works all the time. ripup and reroute is the only hope for success.

Once the edges that block the current route have been identified. nets which pass through this edge must be selected for rip-up. One possibility is to select the shortest net available. The rationale is the same for the shortest-first initial routing: make the smallest modification possible. Another possibility is to rip up the longest net available. since presumably it has the most flexibility for re-routing.

It is also possible to take into account the number of times a net has already been ripped up. If a net has been rerouted a lot of times already. perhaps it would be good to give some of the other nets a chance. On the other hand. if an effort is made to pick the most likely nets to rip up. there is little point in trying all the other nets equally often if the likely ones don't help.

Another possibility. which was used in [52]. is to rip up *all* the nets. and iterate until the no further improvements are made in the maximum density. This has the advantage of allowing us to put off the question. "How low a maximum density is good enough?". since it will continue as long as it can improve this quantity. Also. given the maze routing strategy above. it will never give a net a net a route that is worse than the previous route. The problem of order dependence is also reduced. as can be seen from Table 3.5.

Some alternate strategies that we did not explore are partial ripup [61] and recursive rerouting. In partial ripup. a "weak modification" that pushes only a part of the route aside is made whenever possible. In recursive rerouting. a net would be ripped up. the primary net would be extended only far enough to occupy the space thus freed. and then the second net would be rerouted. This rerouting may induce the ripup and reroute of a third net. and so on. Both these strategies have been used successfully for detailed

Ratio	Multiplier
0.8	4
0.9	8
1.0	16
1.1	32

Table 3.4: Density Ratio Coefficients

switchbox routing. and they might prove useful for global routing.

Termination Criteria

Finally, there must be a way to determine when enough rip-up and reroute has been done. If all the nets are ripped up and rerouted in order. the process can be terminated when no further improvement is achieved. Otherwise, there must be some limit on how many overall ripups are done, or a limit the number of times a particular net can be ripped up. In the latter case. if a net has exceeded its ripup limit, it is locked in place.

Calculating Edge Weights

Before maze routing can be attempted. the density of routing at each block boundary must be translated into a cost for the arc between the corresponding maze nodes. Maze arcs have costs that range from 0 to C_{max}. A cost of ∞ indicates that the arc may not be taken. Instead of blocking off an arc in this manner. a function of the density is used to map from the density to the cost that grows rapidly as the density approaches and exceeds 1. This is

$$cost(d) = C_{max}\alpha(d)d.$$

This has the effect of encouraging lower crossing densities. Table 3.4 gives the values of α for various density ratios.

Results

The numbers in Table 3.5 were generated by running the DES example. using different ordering strategies and enter cost functions. The cost weights for the runs with varying arc weights were taken from Table 3.3. and the constant weights were taken from

Initial Ordering	Arc Weights	Edge Density max / avg	Net Size max / avg
random	constant	0.79 / 0.53	159 / 6.42
random	changes	0.92 / 0.54	157 / 6.41
shortest	constant	0.79 / 0.53	160 / 6.41
shortest	changes	0.88 / 0.54	157 / 6.41
longest	constant	0.81 / 0.54	159 / 6.41
longest	changes	0.88 / 0.54	157 / 6.41

Table 3.5: Global Routing Ordering Strategies

Enter Cost	Arc Factor	Edge Density max / avg	Net Size max / avg
0	0.0	0.89 / 0.54	157 / 6.41
0	0.1	0.79 / 0.53	160 / 6.42
0	0.3	0.79 / 0.54	160 / 6.44
0	0.5	0.79 / 0.53	159 / 6.43
10	0.0	0.88 / 0.54	157 / 6.40
10	0.1	0.79 / 0.53	158 / 6.43
10	0.3	0.79 / 0.53	159 / 6.43
10	0.5	0.79 / 0.53	161 / 6.42
100	0.0	0.88 / 0.54	158 / 6.40
100	0.1	0.79 / 0.53	160 / 6.42
100	0.3	0.79 / 0.53	159 / 6.41
100	0.5	0.79 / 0.53	159 / 6.42
30	0.0	0.92 / 0.54	157 / 6.39
30	0.1	0.79 / 0.54	161 / 6.42
30	0.3	0.81 / 0.53	161 / 6.44
30	0.5	0.89 / 0.61	166 / 6.95
60	0.0	0.89 / 0.55	157 / 6.39
60	0.1	0.81 / 0.53	160 / 6.42
60	0.3	0.79 / 0.53	159 / 6.42
60	0.5	0.79 / 0.54	160 / 6.43

Table 3.6: Constant Values for Enter Cost and Arc Weight

Table 3.6. It is interesting that much better results were obtained when the same enter costs and arc factors were used for all passes. instead of values which depend on which pass this is. This result is different from that obtained by Hong [53]. who found varying weights to be more effective.

If All Else Fails

If no nets can be ripped up at a particular over-congested edge. we must give up and leave the net unrouted. This is a very bad thing. since either a human or a final clean-up stage of the program must route the net individually after the final placement and routing is completed. This sort of human intervention is unacceptable for rapid-turnaround ASIC design. It is time-consuming. error-prone. and quite difficult. Therefore. there must be some sort of last-gasp algorithm that routes the (hopefully) few nets that remain after the final place and route.

Much more information is available after the detailed place and route is completed. In particular. the exact positions of all the blockages caused by both cells and routed nets are known. However. a flat maze route of the entire chip surface is unacceptable. In a chip with a few thousand cells. the maze might be as large as 1000×1000. with a million graph nodes. This is bound to be very space- and time-consuming.

On the other hand. it should be possible to use the detailed information available to formulate a much more accurate set of capacities for global routing. The unrouted nets can then be globally routed and added to the detailed routing of the individual blocks in the route. We have not yet explored this option. however.

Chapter 4

DETAILED PLACEMENT

After the chip has been decomposed into a grid of small leaf problems (or blocks) by means of partitioning and global routing, these problems must be placed and routed. The detailed placement process uses simulated annealing, and the routing is done by means of a switchbox router.

4.1 Ordering the Detailed Placement Problems

The success of the detailed placement and routing process depends strongly on the ordering of the blocks. If none of the neighbors of a block has been placed and routed, all the pins on its borders are floating, and it should be easier to place and route than if some of the pins were fixed, since there are more degrees of freedom. (This may not always be a correct assumption — the move set and the solution space will be bigger and it may take longer to find a solution with as low a cost. However, it does not appear to be the case that constraining the solution by doing adjacent blocks first leads to a better solution.)

The difficulty of each problem is determined by adding up the percentage utilizations of routing capacity on the four sides, and the problems are then done in decreasing order of difficulty. Table 4.1 shows the results of using some different difficulty functions: the one described above, the maximum density of all four sides, and a combination of the sum of the density and percentage of cell area used. The function that yields the best results is the one that incorporates the most information about the difficulty of the problem, which is the sum of the edge densities and the percentage utilization of the cell area.

Ordering Strategy	Circuit Name	Number Blocks	Number Undone	Number Retries
$\sum density$	adder32	28	1	4
max($density$)	adder32	28	0	7
$\sum density + \%area$	adder32	28	0	5
$\sum density$	pr1	64	23	13
max($density$)	pr1	64	17	18
$\sum density + \%area$	pr1	64	17	22
$\sum density$	des	256	62	69
max($density$)	des	256	67	76
$\sum density + \%area$	des	256	51	61

Table 4.1: Ordering Strategies for Detailed Placement

4.2 Adaptive Simulated Annealing

The general simulated annealing algorithm, as described in [35], is as follows.

```
temp = initial_temperature( );
WHILE (outer loop not done) {
    WHILE (inner loop not done) {
        move = select_move( );
        change = cost_change(move);
        IF (random( ) < accept_func(change, temp))
            accept move;
        ELSE
            reject move;
    }
    temp = update_temperature(temp);
}
```

In a simple annealing algorithm, the initial temperature would be a constant, the outer loop would finish either when a pre-set temperature was reached or when no moves were accepted, the inner loop would finish when a certain number of moves per item have been tried. *accept_func* would be $e^{(change/temp)}$, and the temperature would be updated by multiplying the current temperature by a constant, typically between 0.85 and 0.95.

A number of improvements can be made upon this selection. These improvements are described in [57], but most of the constants given below have been empirically determined. The annealing should be started at a temperature where most moves are accepted.

in order to be sure that the solution space is covered evenly. This is done by generating a number of moves and calculating the standard deviation from the initial. random configuration. The initial temperature is then taken to be proportional to this standard deviation. Empirical tests have yielded a value of 3 as a good constant of proportionality: for this value. over 90% of all moves are accepted.

The outer loop stopping criterion is kept the same, that is. when no states are accepted during this pass. Instead of simply exiting the outer loop here. however. the inner loop is repeated one more time. with the temperature equal to zero. This last pass usually finds a few more cost-improving moves, but seldom makes a big improvement.

The inner loop stopping criterion is more interesting. The idea is to run at one temperature until the system has "reached equilibrium". At this point. no further good can be accomplished without reducing the temperature, because any time a significantly better configuration is reached, the temperature is such that it is likely that this minimum will soon be left for another. possibly much higher, configuration. As described in [57]. equilibrium is detected by maintaining a count of how many accepted moves have cost changes that are within a particular window. and if this count exceeds a certain threshhold before the total number of accepted moves exceeds a different threshhold, equilibrium has been reached. Both of these threshhold values depend on the previous value of the standard deviation of the cost. As a limiting case. the inner loop is terminated if a certain number of moves has been accepted. or if a certain number have been tried. Both these limits depend only on the problem size.

The temperature decrease algorithm. like the inner loop stopping criterion. depends on the standard deviation of the costs at the previous temperature point. If the standard deviation was higher than expected. then the temperature should be decreased by a smaller amount. since the previous temperature decrease was too great. On the other hand. if the standard deviation was small. then the previous decrease was too conservative and the current decrease can be made larger. The formula used is $T_{new} = \alpha T_{old}$, where $\alpha = \epsilon^{\lambda T_{old}/\sigma}$. σ is the standard deviation of the cost at the previous time point. and λ is a constant between 0.5 and 1.0. Since a higher value of λ leads to a slower temperature decrease. and, all other things being equal. a better but slower final solution. the value of λ is determined by the *quality*, which is a parameter of the annealing algorithm. Since this formula may lead to very small values of α for high temperatures, an upper limit on the temperature decrease possible is set. For simplicity, the value of λ given above is used as the minimum value for α.

4.3 The Cost Function

The cost function used by the placement algorithm contains a number of terms. They are:

- total net length
- cell overlaps
- external pins that are blocked by cell geometry
- slack between pairs of cells
- slack between cells and external pins
- maximum net density
- net density / net capacity ratio

The last two, which deal with net density, turned out not to be useful, and are thus not calculated in the current implementation. Note that all distance values are calculated separately for the horizontal and vertical directions, and may be weighted differently for each direction.

Net Length

The total net length is simply the sums of the sides of the half-perimeter bounding-box for the net. Note that for some nets, it is possible to ignore either the horizontal length or the vertical length, or both. If a net has external pins on both the left and right edges of the block, no cell or pin move can change the horizontal span of the bounding box. (Figure 4.1.) If it has pins on all four sides, no move can change the bounding box at all. When this is the case bounding box need not be re-calculated.

Overlaps

Cell overlaps occur when parts of two or more cells occupy the same region. Since nothing prevents us from attempting cell moves that cause overlaps, we must weight overlaps highly in the cost function so that even if some do occur in the early stages of annealing, they

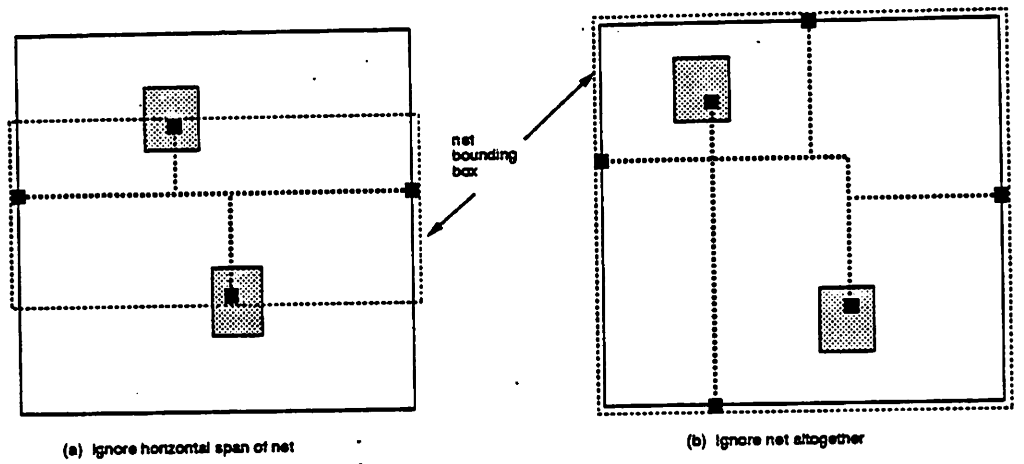

Figure 4.1: Ignoring the X- or Y-spans of a Net

will soon be eliminated as the temperature decreases. The cells may have any rectilinear shape. so we must be prepared to form the intersection of arbitrary Manhattan polygons. In most cases. however. the cells are rectangular. so we special-case polygons with four vertices and avoid the expensive Manhattan intersection calculation.

Pin Blockages

Whenever a cell blockage is within one grid unit of an external pin. that pin is considered *blocked*. Since it is difficult or impossible to route to such a pin. total pin blockages are included as a term in the cost function. For efficiency. two arrays of flags are maintained for each edge: the *pins* array and the *pinblocked* array. Whenever there is a pin in a particular position. the *pins* value for that position is 1. and whenever there is an obstacle that affects a particular position. the *pinblocked* value is 1. These arrays are maintained by the *pin move* routine and the *cell move* routine. respectively. To calculate the total pin blockage factor we merely compare the two arrays.

Cell and Pin Slack

If netlength were the only major consideration. the produced layouts would often be unroutable. The cells would tend to group tightly together, since they are in general more tightly connected to each other than they are to the external pins. In order to prevent this. and to leave adequate room for routing between the cells. the *slack* between cells is taken into account in the cost function. The slack between two cells c_1 and c_2 is defined as $\max(0, minsep - \text{dist}(c_1, c_2))$. The slack value is measured independently in the X- and Y-direction. $minsep$ is currently set at half the width or height of the *basic tiling unit*, the smallest unit from which the chip template can be made without rotation or mirroring. This is approximately the size of an average cell. Thus the total cell slack will be zero roughly when there is a cell's width or height between every two cells.

It also helps the routability of the layout to take note of the positions of the external pins, and push cells away from them. If there are a lot of external pins in one area, a lot of routing room will be required in that area. *Pin slack* is calculated in the same way as cell slack but is between cells and external pins. For every { *cell, extpin* } pair we add $\max\{0, minsep - \text{dist}(cell, extpin)\}$, to the pin slack factor.

This factor has significant effect on the layout. It tends to gather external pins together in one area and open up routing regions for them, which is useful if they are on nets that merely feed through the block. The cells tend to gather in the center of the block, which creates "islands" of cells in a sea of routing (See Figure 4.2).

Another possible way to accomplish this would be to add dummy cells that are attached to nets when there are many nets that want to take the same route. This would be equivalent to measuring slack from the center point of a net, as opposed to from the pins.

Net Density

The use of net density as a term in the cost function was another attempt to use the quantity that we are really trying to minimize directly as the cost function. We tried using two terms based on the density — the maximum density across the block, and the density to capacity ratio, which is defined for the horizontal case as $sum_{x=0}^{max} dens(x)/cap(x)$, where $dens(x)$ is the number of nets that cross the vertical line at x, and $cap(x)$ is the number of free horizontal tracks that cross the same line. The vertical density ratio is defined similarly.

The motivation behind including these terms was the observation that there are cases where minimizing net length and slack do not in fact minimize routability, much like minimizing the total crossing count in mincut partitioning does not minimize the global routing density. The density factors were relatively time-consuming and troublesome to compute and maintain, causing the annealing process to run twice as slow for the same number of moves. The costs obtained by this method were no better than those obtained

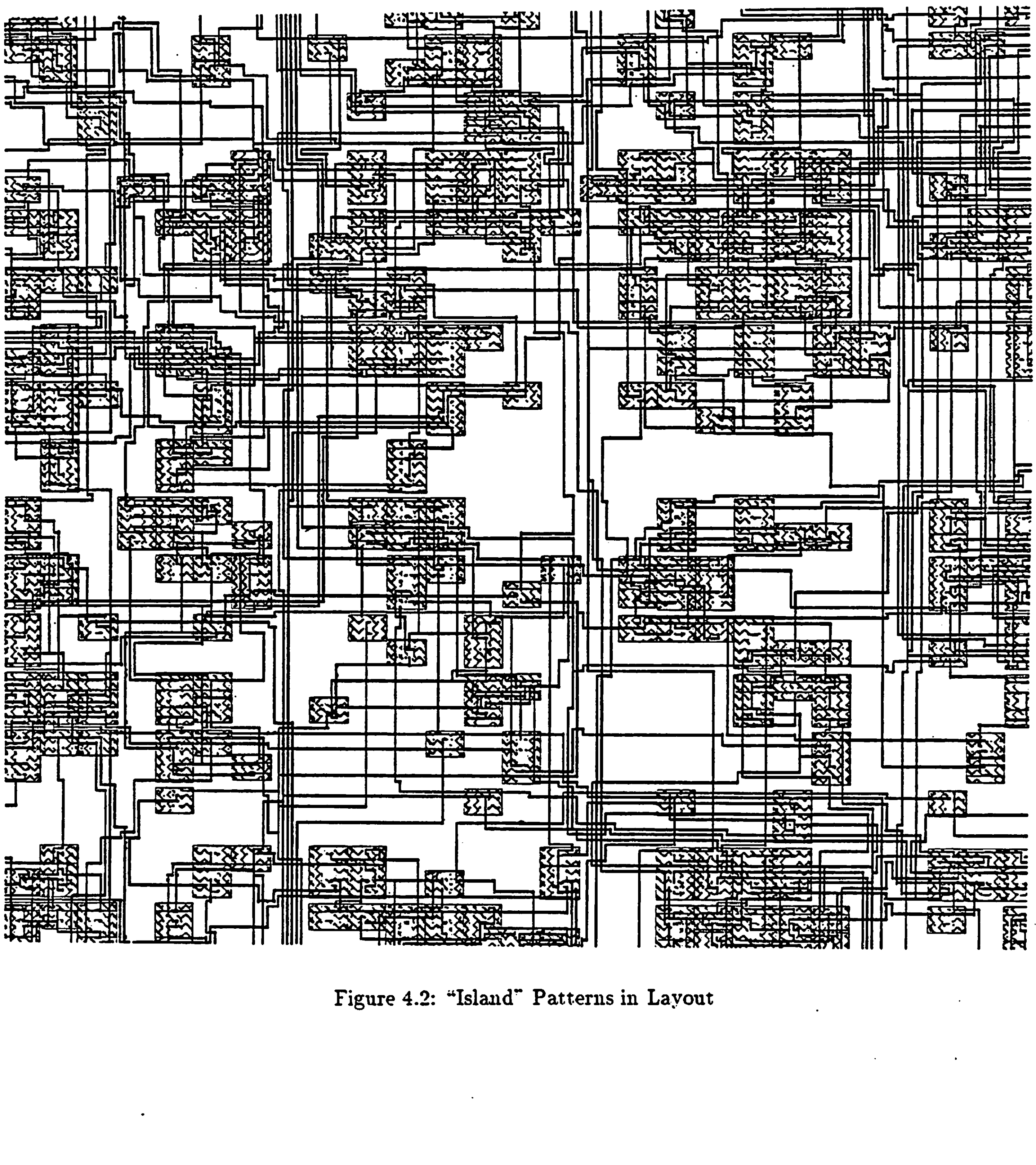

Figure 4.2: "Island" Patterns in Layout

without density factors, which suggests that the cases where minimizing netlength and slack is very far from minimizing density are relatively rare.

4.4 The Move Set

There are three types of moves available during placement. These are cell moves, cell swaps, and pin moves.

Cell Moves

A cell move consists of changing the location of one cell. Each cell can be thought of as being in one *basic tiling unit*, or BTU. Within that BTU, it resides in a particular *slot*, and has a particular *transformation*. Thus its location can be expressed as a $\{BTU, slot, transform\}$ triple.

First, a new BTU is picked for the cell. **Admiral** uses *range limiting* when making this selection, a technique used by simulated annealing algorithms to increase the percentage of accepted moves at low temperatures. The new BTU will be at most distance δ_x horizontally and δ_y vertically from the old one, where

$$\delta_x = width \log_{10}(9\sqrt{(T/T_{init})} + 1).$$

and

$$\delta_y = height \log_{10}(9\sqrt{(T/T_{init})} + 1).$$

T is the current temperature, and T_{init} is the first temperature point used in the annealing. The rationale behind using the logarithm of the temperature is that the temperature is expected to decrease roughly exponentially with time, so the range will decrease linearly.

Once a BTU has been chosen for the cell, a $\{slot, transform\}$ pair within that BTU must be chosen. Each cell has a list of $\{slot, transform\}$ pairs which are valid for it. For instance, in Figure 4.3, the nand2 cell has the following set of $\{slot, transform\}$ pairs.

s0.0	NO_TRANSFORM
s0.1	MIRROR_X
s1.0	NO_TRANSFORM
s1.1	MIRROR_X

A $\{slot, transform\}$ pair is chosen at random for the cell. Note that the leaf placement region is always composed of whole BTU's, so *slot* will exist for that BTU.

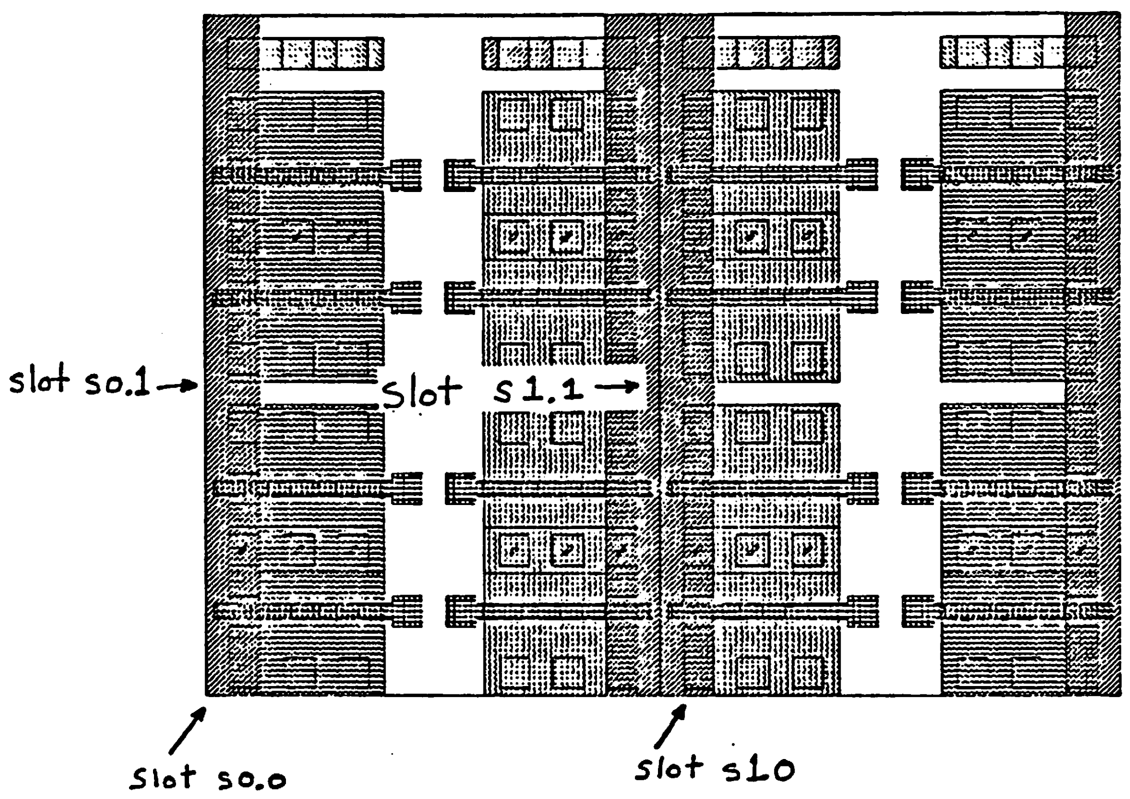

Figure 4.3: Slots Possible for a **nand2** Library Cell

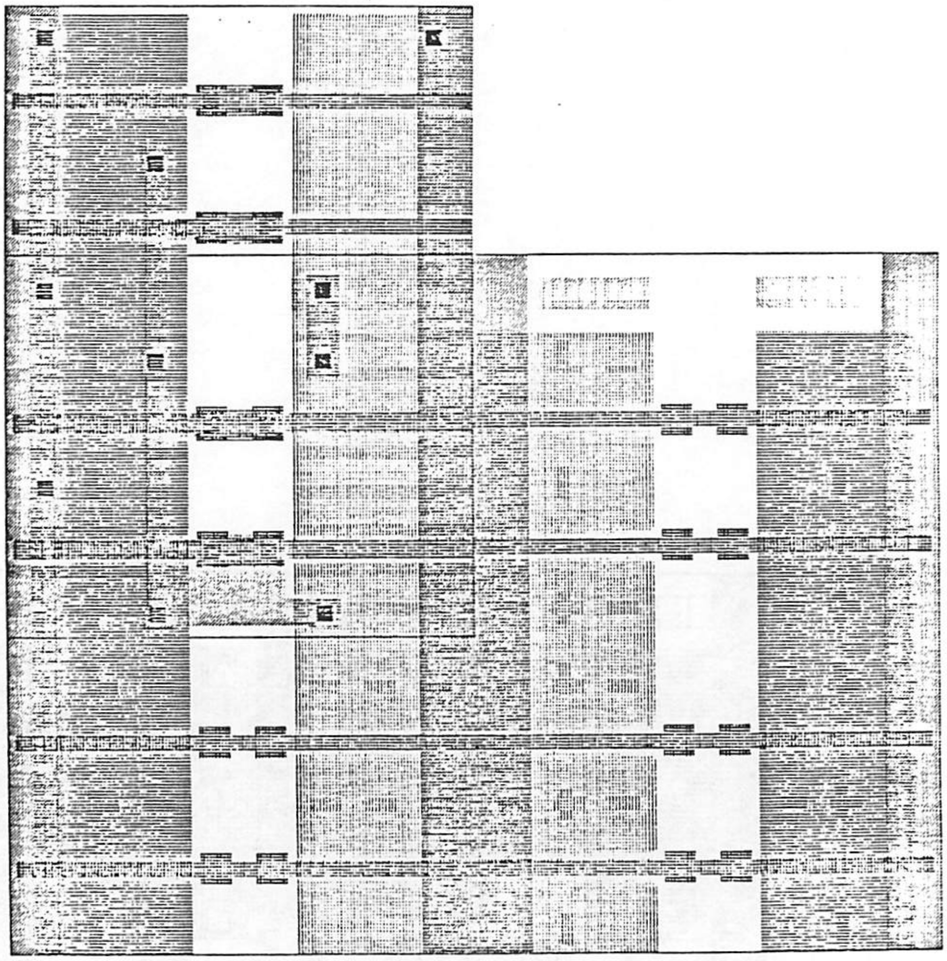

Figure 4.4: A Cell "Spilling Over" Because of the Wrong Slot Choice

However, a cell may spill over the side of the BTU, as in Figure 4.4. In this case, if the cell would go outside of the placement region, the move is discarded and another is tried. Since the sizes of the cells present in a block are carefully considered when the decision whether to terminate the partitioning process is made (as described in Section 2.4), there are never cells that cannot fit somewhere in the placement region.

Cell Swaps

A cell swap is much the same, but instead of one cell two are moved. The BTU's of the cells are interchanged, and if the old slot and transformation used by one cell is valid for the other cell, it is used for that one. If it is not, then a new, valid {*slot, transform*}

Number Cells	Number Ext Pins	Assigned Before	Assigned During	Assigned After
5	14	1312	1011	1172
9	5	2593	2216	2442
14	52	5761	5571	5499
15	35	4602	4422	5043
22	17	3871	3422	3557
23	55	3962	4001	4552
26	67	8841	8256	9825

Table 4.2: Pin Assignment Algorithm Results

pair is selected at random. Unlike cell moves. cell swaps do not use temperature-dependent range limits. It is not clear whether using them would improve the placement quality.

Pin Moves

The third type of move that may be generated is the *pin move*. which consists of moving a floating pin from one position on its edge to another position on the same edge. The distance a pin may move is controlled by the same range-limiting technique that is used by pin moves. An invalid location for a pin will never be generated (e.g. a METAL-1 pin in the same location as a power or ground rail). two pins will never be placed on top of each other. since this is much easier to enforce than a prohibition against cell overlaps. and a pin will never be placed within two positions of a corner. This is important because some detailed routers. including Mighty. have severe problems with congested corners.

The rationale behind moving pins as part of the simulated annealing process is as follows. There are three choices for determining pin locations. First. they can be determined beforehand. perhaps by means of a linear assignment algorithm that looks ahead to the locations of pins on the same net but in different blocks. Second. they can be determined concurrently with the locations of the cells. which is the approach we use. Third. they can be determined after the placement is finished. perhaps as part of the routing process.

Results for these three pin-assignment techniques are given in Table 4.2. Although the concurrent approach is much more expensive. since there are many more moves. the results are significantly better.

It is possible to control the probabilities of generating the different types of cell

Cell Move	Cell Swap	Pin Move	Total Time	Moves Tried	Moves Accepted	Final Cost
1	1	1	67	12993	2323	5587
1	0	1	43	15038	3605	5274
1	1	2	58	13385	3047	5415
1	2	1	77	12799	2513	5303
2	1	1	66	12575	2137	5186
2	2	1	62	10263	1707	5702

Table 4.3: Results for Different Weight Sets

moves. Table 4.3 gives some statistics for different weight sets. Note that the probabilities are not dynamically adjusted as a function of temperature. Adjustment for temperature is made to a large extent by using range limiting, although it is possible that pin moves might have different optimal probabilities relative to cell moves and swaps at different temperatures.

Chapter 5

DETAILED ROUTING

After a detailed placement has been obtained for a block. the block must be routed. **Admiral** currently uses the **Mighty**[62.61.63] channel and switchbox router to accomplish this.

5.1 Routing with Mighty

The **Mighty** router is a two-level router that assumes a routing grid. It allows pins to be fixed anywhere within the routing region and the top and bottom edges. and it allows them to float on the left and right edges. although this feature seems not to work with many internal blockages. The information we must provide to **Mighty** consists of:

- a rectangular routing area
- a set of blockages within the area
- a list of pins within the routing area and on the fixed edges
- a list of nets. possibly with criticalities

These are all straightforward to extract from our representation. All of the pins are fixed on the periphery of the routing region. with pins on the top and bottom sides of the block in METAL-1 and pins on the left and right sides in METAL-2. The blockages are derived from the cells and the power and ground rails. We write an ascii input file for **Mighty**. run the program. and read in the routed net information.

Figure 5.1 shows an unrouted block and the same block after the routing has been completed. The nets in the first picture are drawn as "spiders". with lines connecting each

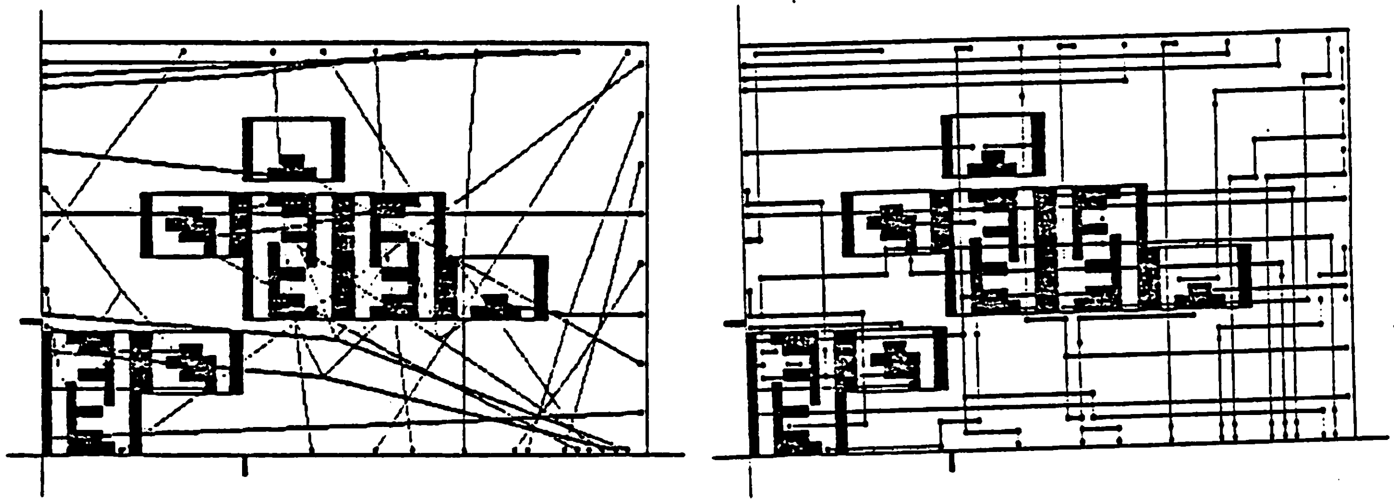

Figure 5.1: Unrouted and Routed Blocks

pin to the midpoint of the bounding box. In the second picture they are drawn as output by **Mighty**. This picture was drawn by the graphical debugging option to **Admiral** (the -g flag).

5.2 If Mighty Fails

In the event that **Mighty** fails to route the block. the quality factor for the annealing is increased according to the following formula:

$$Q_{new} = 1 - 0.75 + Q_{old}0.75$$

Additionally. the pin slack cost weight is multiplied by the quality factor. The assumption made is that if the routing failed. it is probably because too much weight was given to minimizing net length. at the expense of routing area. Table 5.1 shows the number of blocks completed at each try for some examples.

We cannot extract much more information from **Mighty** than the fact that it failed. **Mighty** tells us which nets it is ripping up as it does it. but this can't help us determine which nets. if any. would. if removed from the problem. allow the block to be routed.

If. after several tries (currently this number is 6) the block is still unroutable. the program gives up on this block. Another approach that we tried was ripping up all the

Circuit Name	Number Blocks	1st Try	2nd Try	3rd Try	4th Try	5th Try	6th Try	Not Done
adder32	32	27	4	1	—	—	—	—
pr1	64	52	7	2	3	—	—	—
pr2	228	201	10	7	5	4	1	—
des	256	203	41	11	1	—	—	—
mult32	348	264	17	19	26	18	2	2
mult32x6	2012	1029	251	54	91	5	3	579

Table 5.1: Number of Detailed Layouts Completed at Each Iteration

Num Cells	Ext Pins	All Sides Free	Three Sides Free	Two Sides Free	One Side Free	No Sides Free
17	41	4633	5039	5193	5416	6516
20	42	5587	6511	6818	6135	7145
28	38	10401	8925	9380	9882	10264
4	20	1229	1225	1254	1280	1084
5	19	1396	1276	1204	1393	1401
6	27	1651	2072	2159	1976	2217

Table 5.2: Costs for Blocks More or Less Constrained

neighboring blocks (if any were previously placed and routed) and placing the current block with no constraints. It was expected that this would allow more blocks to be completed by removing the dependence upon the order of the problems. but this did not occur. It seems that if there are a lot of difficult problems. they tend to be close together. and at least a few of them will have to be done with external constraints provided by the others.

Table 5.2 shows some statistics for the placement costs obtained for some blocks when they are placed and routed at different points in the sequence. and have different numbers of edges constrained. Note that the costs of the blocks become higher as the placements become more constrained. but the differences are usually within 10%. More important than the placement cost. however. is the fact that very often blocks may be unroutable when placed with constraints. while they become routable when placed unconstrained.

5.3 An Ideal Detailed Router for Mariner

The **Mighty** router has a number of shortcomings which make it less than ideal for use with a Sea of Gates layout system.

First, it is capable of routing only two layers of metal. While most available Sea of Gates processes provide only two layers of customizable metal, three or more layers will certainly be widely available in the near future. Several vendors [45] already provide a third layer of metal.

Although in N-layer metal technologies, where $N \geq 3$, the first layer of metal is likely to be used mainly by intra-cell routing and the others are likely to be used mainly in inter-cell routing, a detailed router should be able to make use of free area in METAL-1 and accommodate blockages in the other layers.

Multi-layer channel routers have been written [6], and it should be a straightforward extension to a switchbox routing algorithm that is based on maze routing, such as **Mighty**'s, to handle more than two layers.

Second, **Mighty** allows pins to float on only two sides of the switchbox. We were not able to obtain good results using this feature, so we had to fix the positions of all the external pins. Since positions for these pins anyway are calculated as part of the placement process, this was not a problem. However, there were many cases where the placement algorithm generated a placement that the router could not route, but it was clear that if some of the external pins could have been moved, the routing would have completed.

Our ideal router should be able to handle floating pins on all four sides, and generate routes that are as good as or better than those achieved when any of the pins are fixed. In practice, a good guess as to the final pin positions might help the router a great deal, but if the positions generated by the placement were not entirely fixed, the pin assignment technique could be relaxed somewhat, as described in Chapter 4, leading to reduced runtimes.

Also, for simplicity we assign all pins on horizontal edges to METAL-1, and all pins on vertical edges to METAL-2. The METAL-2 pins must be in that layer, since in our library METAL-1 is always used for power and ground busses on the edges of the blocks, but the METAL-1 pins on the horizontal edges do not have to be in that layer. In fact, Mighty breaks its wiring model quite often and routes vertical nets in METAL-2. It sometimes occurs that a net which is being routed vertically in METAL-2 reaches a block boundary, goes down to METAL-1. makes the transition to the next block. and then continues again in METAL-2. This unnecessary use of vias could be avoided if the router could decide for itself which metal layer to use for these floating external pins.

The third problem with **Mighty** is that it (like most routers) assumes that it can place vias anywhere it likes. Because of the way the **Mariner** template was designed. there are some places where vias cannot be placed. due to design rule constraints. Because of this. the layouts currently produced by **Mariner** are not design-rule correct. There are two solutions to this problem: either modify the template to make it possible to put a via at every grid location. or add enough intelligence to the router to enable it to avoid putting vias at these forbidden locations. The first solution. although much easier. would require expanding the transistors by 2 μm per transistor. which is undesirable.

The last consideration is a general one which is not due to any specific characteristic of Sea of Gates. Most of the cells in the library have either equivalent or permutable pins. Two pins are equivalent if a net could be connected to either of them without any electrical difference. Since the pins are usually in the middle of the cell. and are connected to by running a wire over the cell to the pin location. any position on the wires inside the cell implementing the internal net corresponding to the pin would be equally good for a connection. (Of course. there may be positions that are impossible to place a via. and some positions may be bad because of second-order effects like crosstalk. but these effects are usually not considered by digital logic routers anyway.)

A set $\{p_0. p_1. \dots p_n\}$ is permutable if the nets $\{n_0. n_1. \dots n_n\}$ connected to these pins may be permuted arbitrarily without affecting the logic function of the gate. An example of permutable pins are the input pins of an AND gate. If a router is capable of permuting the nets attached to a set of permutable pins. it can often achieve a better route. Of course. our detailed placement is carried out using the exact positions of the pins. so very often the best permutation of the pins for a given placement will be the one used already.

Dealing with floating. equivalent. and permutable pins is easier for most maze routers than for other types of routers because the basic idea is to expand until the destination pin is reached. If the destination pin is replaced by a set of pins. or a range of possible pin locations. the same general algorithm can be used. stopping when any one of the alternatives is reached.

The **Hero** router [42]. currently under development. should address some of these problems. Plans for future enhancements include an arbitrary number of layers and floating pins. The **CODAR** router [69] also contains many of the features mentioned here.

Chapter 6

STATUS, CONCLUSIONS, AND FUTURE WORK

6.1 Current Status

The **Mariner** system is currently running, and we have obtained results for a variety of small to medium size problems. We have not been able to get acceptable results for examples larger than about 5,000 cells, for reasons which will be explained in the Conclusions section.

Table 6.1 shows some statistics for a number of examples, which include some industrial netlists (pr1, pr2, hughes), some interesting netlists created by the logic synthesis tool **MIS** (sm1, des), some simple structures also created by **MIS** (adder8, adder16, adder32, mult32), and one large netlist (mult32x6) which can be placed by **Admiral** but not routed. The statistics available are the numbers of cells, nets, cell pins, and distinct cell types in the netlist, the total netlength in millimeters, the maximum attainable transistor utilization percentage, and the "parallelism factor" observed for this netlist (see Section 3.2). Incomplete results are given for some examples, as these netlists are too large for **Admiral** to lay out. They were run in "placement only" mode, and the useful numbers are the runtimes for the placement phases. For these examples, the total netlength given is the total half-perimeter bounding-box length.

Table 6.2 shows the runtimes for these examples, broken down by the various phases of the algorithm. As expected, the global placement phase is roughly $O(n \log n)$. The

Circuit Name	Num Cells	Cell Terms	Num Pads	Num Masters	Total Pins	Net Length	Trans Util	Parallel Factor
pr1	833	6791	0	7	6153	557	44 %	0.41
pr2	3014	25446	0	7	26615	4552.	42 %	0.52
hughes	25917	201746	0	16	—	23051	58 %	—
des	2069	16881	20	15	21532	3245	42 %	0.62
sm1	627	5074	34	11	4017	440	46 %	0.58
adder8	62	490	214	5	311	31	59 %	0.76
adder16	183	1433	48	10	1104	89	50 %	0.70
adder32	376	2943	96	12	4061	155	35 %	0.74
mult32	5971	49625	96	13	—	11651	51 %	—
mult32x6	35826	321278	96	13	—	38694	43 %	—

Table 6.1: Final Results

Circuit Name	Input Time	Global Place	Global Route	Detailed Place	Detailed Route	Output Time	Total Time
pr1	0:47	22:33	7:18	16:47	8:41	2:28	58:34
pr2	4:18	3:51:13	1:19:12	2:11:19	1:10:42	17:19	8:54:03
hughes	20:51	13:57:11	—	11:33:20	—	30:11	26:21:33
des	3:13	53:19	25:00	2:20:41	27:17	37:15	4:46:45
sm1	0:44	1:12:56	30:10	10:18	4:40	3:51	2:2:39
adder8	0:04	0:12	0:15	0:30	0:10	0:07	1:18
adder16	0:15	0:34	0:17	2:21	1:11	0:26	5:04
adder32	0:34	2:10	1:03	3:19	4:13	1:10	12:29
mult32	7:17	30:11	—	2:55:01	—	10:01	3:42:30
mult32x6	30:44	15:19:51	—	13:22:10	—	35:15	29:48:00

Table 6.2: Runtime

Circuit Name	25 % Util	35 % Util	45 % Util	55 % Util	65 % Util	75 % Util
pr1	100	100	100	83	50	12
pr2	100	100	96	96	37	30
adder8	100	100	100	100	100	75
adder16	100	100	100	100	100	50
adder32	100	100	100	100	75	12

Table 6.3: Percent Utilization vs Percent Completion

detailed placement and routing phases scale linearly. since the complexity of the problems does not increase greatly with chip size. The global routing phase. however. seems to grow with complexity of O)(n^2). although for these examples the size is small enough that the maze routing does not dominate the runtime.

Table 6.3 shows the problem completion percentages obtained when the target transistor utilization percentage is varied. As expected. decreasing the transistor utilization increases the completion percentage. The maximum utilization value decreases somewhat as the problem size increases. but not dramatically.

Pictures of the chips adder32. des. and hughes are shown in Figures 6.1. 6.2. and 6.3 respectively.

6.2 Conclusions and Future Work

The work described in this report has shown the basic viability of the Mariner approach to Sea of Gates layout. However. it has also made clear the need for more work in the areas of global routing. detailed routing. and routing area estimation for Sea of Gates.

The highest priority is the development of good global routing algorithms for Sea of Gates. The maze routing algorithm currently used is adequate for small and medium-sized problems. but fails for netlists larger than about 10K cells. Although our problem is different from the typical global routing problem. in that we cannot determine exact capacities since the detailed placement has not been completed. recent gate-array global routing algorithms [44] should prove helpful.

As was mentioned in Chapter 2. there are algorithms which use a hybrid approach to partitioning [55]. The first stage consists of a clustering algorithm that groups tightly-

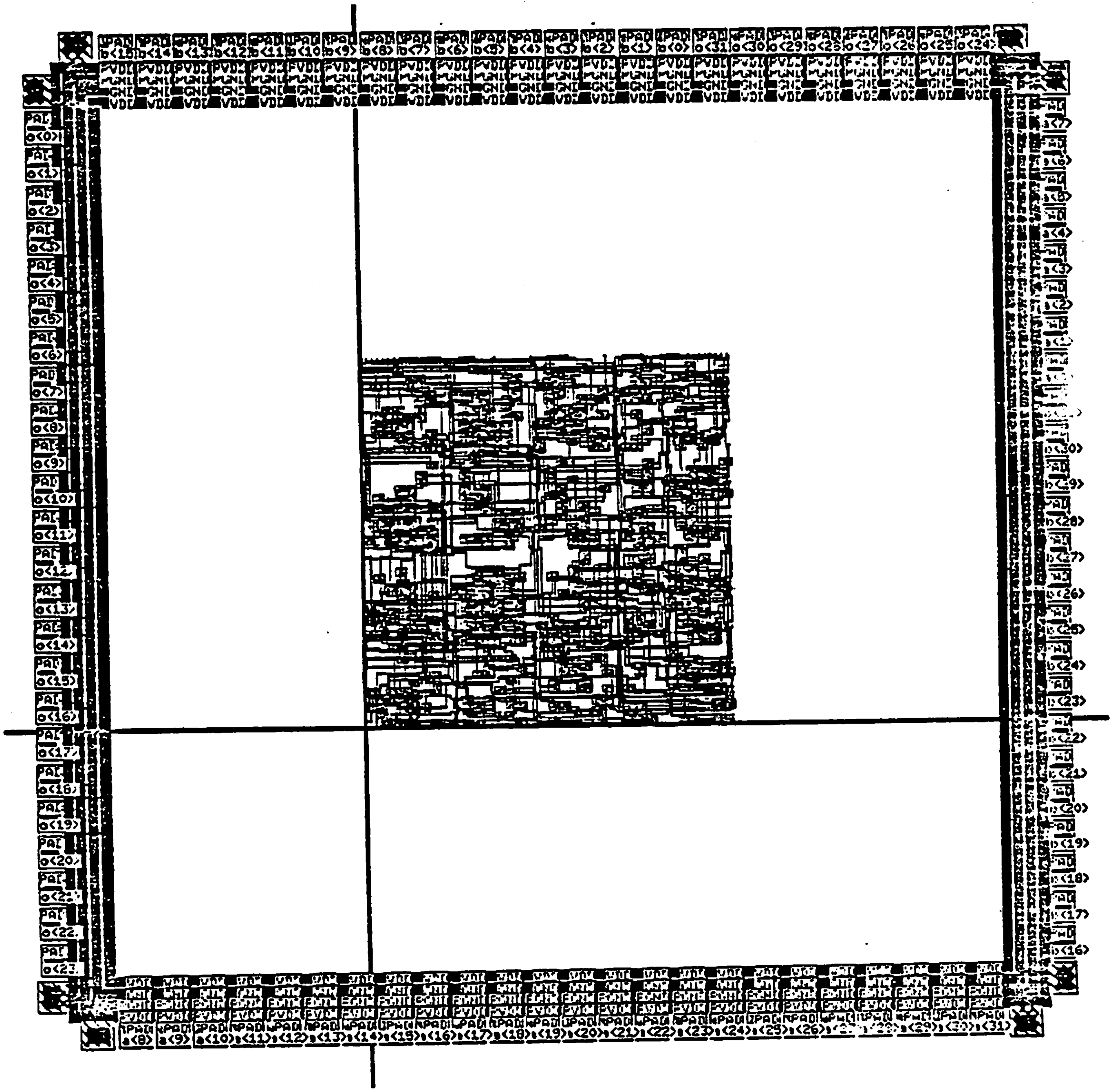

Figure 6.1: The Adder32 Chip

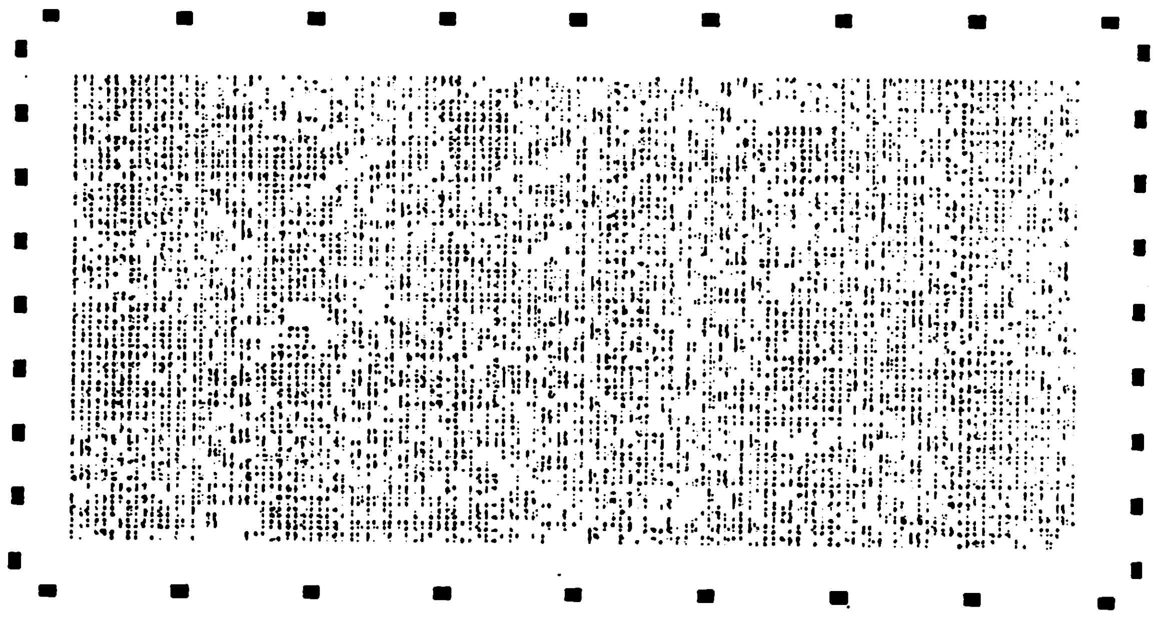

Figure 6.2: The DES Chip

connected cells together, which greatly reduces the size of the problem given to the next stage, which consists of some form of mincut partitioning. This approach has been shown to be promising, and it might prove worthwhile to implement it in **Admiral**.

A detailed router with more capabilities, as described in Section 5.3, would also be helpful, although not essential. In a sense, improvements in detailed placement and routing only lead to a linear improvement in quality or speed, since the leaf problems are always the same size, but improvements to the global placement and routing can have a much higher payoff.

The routing area estimation model is based on empirical experiments done many years ago with different technologies and layout algorithms. We have more layers of metal, different design styles, and more effective algorithms, and it is likely that the formulas obtained by Donath et. al. [17,25] can be improved for our problem by doing more experiments and developing a more Sea of Gates-specific model.

The **Mariner** layout tools have only been tested with one library, the one described in [41]. We have another library [54] which is based on the Siemens transistor template [27], that we have yet to test. Comparisons should be made between the two as to ease of layout, and a comprehensive set of figures should be assembled which compare results obtained with different parameters such as number of feedthroughs for both templates.

Finally, the **Mariner** system needs more backtracking heuristics. Other systems can make mistakes and produce sub-optimal results, but if **Mariner** makes mistakes a

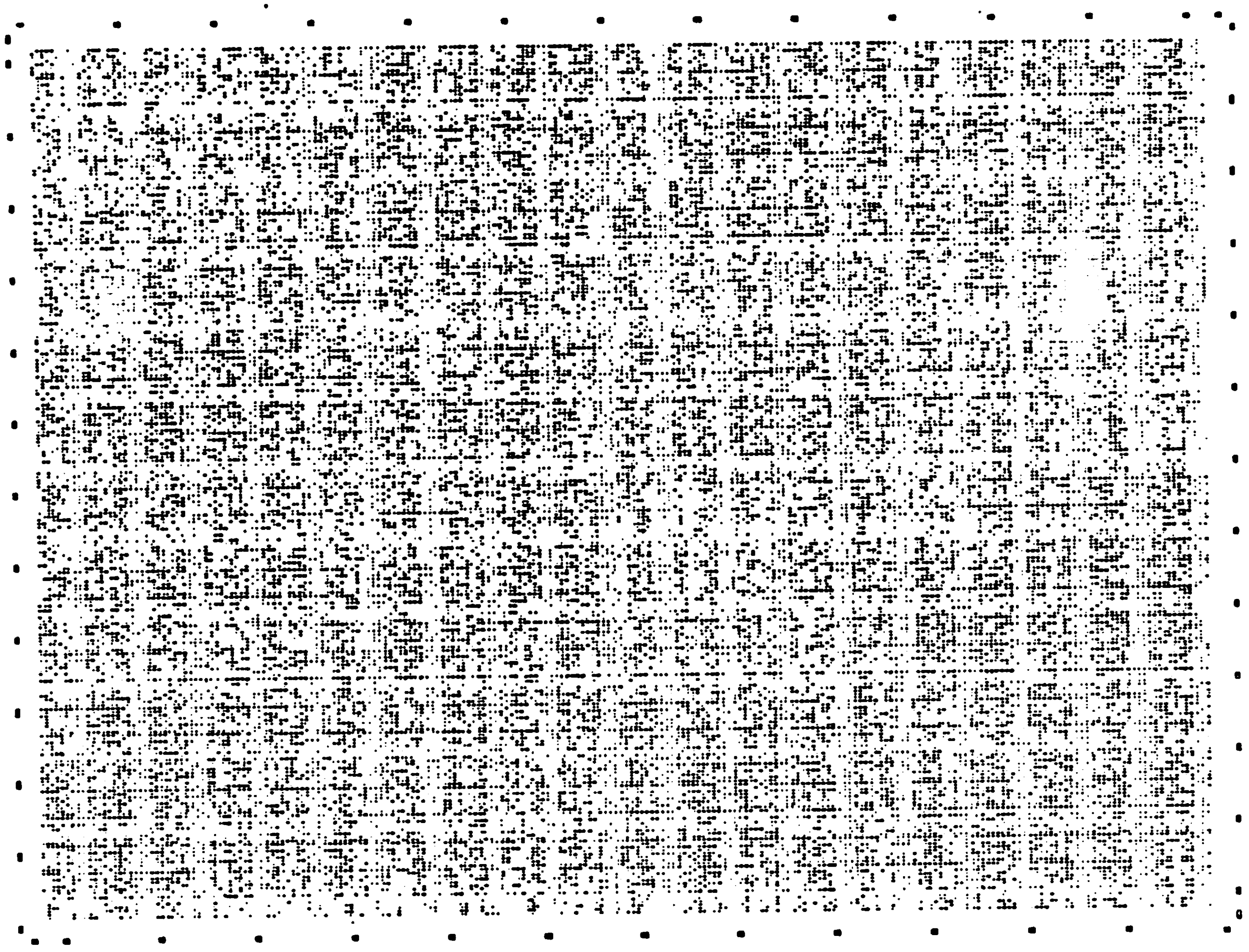

Figure 6.3: The Hughes Chip

correct layout is not produced. In many cases, when a decision is made, it has to be right the first time. We can't easily re-partition on a high level or globally re-route because of lower-level problems, because we would be throwing away too much work for too little gain. If we had more algorithms that allowed **Mariner** to recover from bad initial decisions, we could be more confident that it would generate a working final layout, and we could be less conservative, knowing that if we made a mistake, it would have a good chance of being corrected later. For instance, the "last-gasp" post-placement global routing described in Section 3.3 should be implemented, since it is very difficult to get 100% routing completion the first time.

Appendix A

MANUAL PAGES FOR THE MARINER PROGRAMS

NAME

admiral – Sea of Gates placement program

SYNOPSIS

admiral [options] [-o outputcell[:view]] cell[:view]

DESCRIPTION

Admiral is a placement program – it reads an oct view with unplaced instances and gives them positions which are constent with the Sea of Gates grid, and which should yield good routability and total netlength. The default input view name is "unplaced", and the default output view name is "placed". The basic strategy is a top-down mincut quadrasection phase followed by a series of small placements by means of simulated annealing. An intermediate output facet, "clustered", may be used as input or output.

The options are:

-af number
: Add some "area fudge" to the sizes of the cells for partitioning purposes. When doing the mincut partition, each cell will seem to be larger by this many "regions". A "region" roughly corresponds to one transistor. This option may be necessary if a placement is rather tight and the user wishes the slack to be distributed more evenly than the partitioning is likely to do. Of course, the real cell area plus the total fudge must not exceed the area available for placement.

-ag file Use "file" as the prefix for simulated annealing graph files (see **adaptive**(3CAD)). Note that these files will be overwritten by each simulated annealing step.

-ap name val
: Provide a simulated annealing parameter. Currently there aren't any.

-pa alg Use the partitioning algorithm *alg* for the mincut partititioning. Some available algorithms are "quad", "klfm", "anneal", and "adapt". The default is "quad" which yields good results quickly. "adapt" may yield better results but will take much longer. See **part**(3CAD) for more details.

-po Do the mincut partition only, and exit after writing a "clustered" view.

-pp name val
: Provide a partitioning parameter. Currently there aren't any.

-sl Run in slave mode. This is used when **admiral** is running in multi-processing mode and should not be given by the user.

-spr file Save partition data in file and exit. This file may be used as input to the testpart program for experimentation with partitioning algorithms. See **part**(3CAD).

-ss file Keep statistics in the named file. Currently this does't do anything.

-t template
: Use the given template for technology information. The default is ~cad/lib/technology/mariner/cells/template.

-tf Pre-process the oct facet and write the relevant data into a file, and then read the data into **admiral**. Upon output, write the new locations into the file and then update the oct facet. This option may be necessary for very large oct facets.

-d Debug mode.

-cf filename
: Use the given file for input and output instead of the oct facet.

-g Graphics mode. Draw a map of the partition after the mincut phase is finished, and show the progress of each annealing step graphically. This only works if you are using X.

-mp Run in multiprocessor mode. This will cause the annealing problems to be distributed to a number of workstations, as specified in the *host file*. This is currently not working.

-ms maxsize

Use maxsize as the maximum size for an annealing problem, or the size below which the mincut partitioning will stop. The default is 25. A larger value will yield better results but cause the program to run more slowly.

-pn Paranoid mode. Do lots of checks for things that can't happen.

-q quality
Specify the "quality" of the simulated annealing. This should be a number between 0 and 1. It's not clear it has any effect.

-v Verbose mode.

-wc Write out an intermediate "clustered" view after the mincut partitioning phase.

-wp freq
Write "partially-placed" views every *freq* annealing steps. Currently unimplemented.

SEE ALSO

adaptive(3CAD), checkplace(1CAD), hydra(3CAD), mariner(1CAD), mpadp(1CAD), part(3CAD), pgroute(1CAD)

AUTHOR

Wayne A. Christopher

BUGS

"Area fudge" is a hack.

The -t option should be replaced by determining the template from the cells in the chip.

The multiprocessing stuff doesn't work at the moment.

NAME
checkplace – check the quality of a placement

SYNOPSIS
checkplace cell[:view]

DESCRIPTION
Checkplace checks for overlaps in a placement, calculates the percentage of the chip area used by cells, and calculates the total and average netlength (half-perimeter) and the standard deviation of the netlength.

It obtains the bounding box of the instance by examining the geometry on the "PLACE" layer in the interface facet of the master. This is necessary because the mariner library cells look bigger than they actually are.

AUTHOR
Wayne A. Christopher

NAME

mflatten – Mariner flatten program

SYNOPSIS

mflatten [**–d**] [**–o** outcell[:view]] cell:view

DESCRIPTION

Mflatten flattens an instance hierarchy to one level. Its function is a subset of that of **octflat**(1CAD), but it is much more careful about memory usage, so it can be used for large (>20K) netlists.

The **-d** option enables debugging output.

SEE ALSO

octflat(1CAD), mariner(1CAD)

AUTHOR

Wayne A. Christopher

NAME

mpadp – Mariner pad placement program

SYNOPSIS

mpadp [–p padp_file] [–v] [–o outcell[:view] cell[:view]

DESCRIPTION

Mpadp arranges the pads in the oct facet in accordance with the information in the specified pad placement file. The default *padp_file* name is *cell*.padp. The -v option causes verbosity.

The signal pads must be present in the facet. Power and ground pads are added as needed.

PAD PLACEMENT FILE

The pad placement file contains lines which specify various parameters of the pad placement. They are as follows.

cell *name cell:view* **;**

A cell is defined with the given name. Typically this will be used to define power and ground cells, e. g,

```
cell VDD /cad/lib/technology/mariner/cells/pads/vddpad:physical ;
```

corner *cell:view* **;**

This specifies the cell to place in the corners of the chip.

bottom *cell-spec ... ;*
right *cell-spec ... ;*
top *cell-spec ... ;*
left *cell-spec ... ;*

These lines specify which pads are to go along the sides of the chip. The direction is counter-clockwise. Each *cell-spec* may be the name of a cell as previously defined by a cell statement, or the name of a formal terminal in the chip. In the former case, a pad of the appropriate type is created and placed, and in the second case the pad corresponding to the formal terminal is located and moved to the correct place. Names of formal terminals must be enclosed in quotes. An example is

```
bottom "a<0>" "a<1>" VDD ;
```

ring *net-name term-name layer.width* **;**

A ring is created around the outside of the chip which links up the terminals on adjacent pads with the name *term-name*. The paths linking adjacent terminals are contained in a net called *net-name*, and are on the layer *layer* and have width *width* (in oct units). For example,

```
ring vdd-ring VDD MET1 1200 ;
```

placebox *width height* **;**

A box is drawn on the "PLACE" layer of the interface facet of the cell, which denotes the area available for placement. The pads are constrained to lie outside of this box. The lower left corner of the box is at (0, 0). The width and height are in oct units.

margin *xmargin ymargin* **;**

xmargin and *ymargin* oct units are left between the inner side of the pads and the place box in the x- and y-directions, respectively. This area should be enough for routing to the pads.

OCT POLICY

All pads must be designed so that they will go on the bottom of the chip with no rotation, i. e, they must face up. The corner cell must go in the lower left corner with no rotation.

SEE ALSO

padp(1CAD), mariner(1CAD)

AUTHOR

Wayne A. Christopher

BUGS

This program duplicates the function of the program padp, although it uses a different mechanism. These programs should be merged.

This information should be specified via oct.

NAME

pgroute – do power and ground routing for Sea of Gates

SYNOPSIS

pgroute [**–v**] [**–o** outcell[:view]] cell[:view]

DESCRIPTION

Pgroute wires up the power and ground columns in a placed view. The default input view is "placed" and the default output view is "prerouted". The -v flag causes verbosity.

SEE ALSO

mariner(1CAD), admiral(1CAD)

AUTHOR

Wayne A. Christopher

NAME

regen – re-create a Sea of Gates library for a new template

SYNOPSIS

regen [–o old_template] [–n new_template] [–d new_directory] cell:view ...

DESCRIPTION

Regen maps a cell library from one *template* (basic transistor structure) to another. The template that is currently used by the cells should be given as *old_template*, the one desired for the new cells should be *new_template*, and the directory that the new cells should be placed should be given as *new_directory*. *Regen* will then re-create each of the specified cells with the new template instead of the old one. If the old cell name was .../cellname:viewname, the new name will be newdir/cellname:viewname (i.e, the leading components of the pathname will be stripped off).

The cells and the templates should conform to the Sea of Gates symbolic policy (as well as the general *oct* symbolic policy). The program does the mapping by building up a symbolic grid for both templates by looking at the positions of the terminals on the template, with the assumption that any terminal will fall on grid lines in the X- and Y-direction, and any grid line will have at least one terminal falling on it. Then it maps the wiring in the old cell onto the grid for the old template and from there maps it to the grid for the new template. It then creates the new cell, makes the appropriate number of copies of the template, copies the important properties, and creates the wiring.

The templates must each have a property called "CENTER", which tells *regen* where the middle of the transistors are. If the new template has more tracks on the outside of the transistors, the grid numbers of the locations near the center should stay fixed.

RESTRICTIONS

The old template and the new template must have the same number of transistors per block and blocks per plug, and must have power and ground routed in the same layer. Otherwise the topology of the cell would be different and the automatic mapping wouldn't make sense.

AUTHOR

Wayne Christopher (faustus@ic.Berkeley.EDU)

SEE ALSO

/cad/doc/mariner/*, ckplace(1), stitch(1)

BUGS

NAME

skipper – A sea-of-gates template generator program

SYNOPSIS

skipper

DESCRIPTION

Skipper is an interactive program which generates sea-of-gates templates for both the basic tiling unit (BTU), (the smallest repetitive unit that doesn't need to be rotated to create a full array), and/or a full chip size array with I/O pads, corner, and spacer pads. The architecture of the BTU has n-, n-, p-, p-type, (NNPP) transistors with power and ground buses that run over the diffusion of the transistors and are shared by adjacent columns.

In making the BTU template, skipper allows the user to decide the N and P transistor sizes, the number of metal1 tracks between power and ground, power and ground widths, and the number of transistor pairs between substrate contacts. The user is prompted for all variables over which he/she has control and for each is provided a reasonable default which either has been calculated from previously given information or is the current reasonable default.

HOW IT WORKS

First Skipper asks if you want to make a new template for the basic tiling unit, if the answer is yes it queries the user for needed information. As it asks questions about the template it also asks some things about the chip size template. This is done so that default parameters can be calculated as closely as possible to what will be needed for the chip size template. If the user doesn't want to make a new BTU he/she can give the name of an already existing one and skipper will use that to create a chip size template, warning if parameters such as power and ground widths are insufficient.

HOW TO RUN SKIPPER AND LOOK AT THE RESULTS

The best way to become familiar with what skipper does is to run it. First time users can easily become familiar with skipper by typing returns for all of the questions so that the template and chip produced will be the result of the default parameters. The default BTU template can be viewed using vem by opening the cell template:physical. The default chip size template is chip:unplaced.

SEE ALSO

vem(1), oct(3), admiral(1)

AUTHORS

Lorraine Layer, Wayne Christopher

BUGS

Send bug reports to "mariner@eros".

NAME

adaptiveAnneal() – adaptive simulated annealing

SYNOPSIS

```
#include "adaptive.h"

double
adaptiveAnneal(params, config, probsize, maxmoves, flags, gfile, quality)
        adaptParams *params;
        adaptConfig *config;
        int probsize;
        int maxmoves;
        long flags;
        char *gfile;
        double quality;

typedef struct adaptParams {
        adaptMove *(*generateMove)();
        double (*calcCost)();
        double (*costChange)();
        void (*makeMove)();
        void (*disposeMove)();
        void (*loopFunc)();
} adaptParams;

typedef struct adaptStats {
        int time;
        double temp;
        double cost;
        int numacc;
        int numrej;
        int numbad;
        double avgcost;
        double sigma;
        double maxcost;
        double mincost;
        double maxchange;
} adaptStats;


adaptMove *
generateMove(conf, temp, inittemp)
        adaptConfig *conf;
        double temp;
        double inittemp;

double
calcCost(conf)
        adaptConfig *conf;

double
calcChange(conf, move)
        adaptConfig *conf;
        adaptMove *move;
```

```
void
makeMove(conf, move)
        adaptConfig *conf;
        adaptMove *move;

void
disposeMove(move)
        adaptMove *move;

void
loopFunc(conf, stats)
        adaptConfig *conf;
        adaptStats *stats;
```

LINKING

cc *[flags] files [libraries]* **~cad/lib/adaptive.a -lm**

DESCRIPTION

The adaptive package provides an easy-to-use adaptive simulated annealing framework usable for a variety of problems. The cooling schedule and the termination and inner loop criteria are determined by the package based on the cost function. For a theoretical description of the technique, see "Probabilistic Hill Climbing Algorithms: Properties and Applications", by Fabio Romeo.

The **params** argument contains pointers to all the problem-specific functions. **generateMove** is called to generate a random move, given a cofiguration. The current temperature and the initial temperature are provided in case the function wants to do range-limiting. The current temperature may be ADAPT_INF_TEMP, in which case the function should return an entirely random move. (This is used to determine a good initial temperature.) Note that the annealing code will only use one move at a time so the return value of **generateMove** may be a pointer to a static variable.

The **calcCost** function should return the cost of a configuration. **calcChange** should return the cost of the configuration after the move is made. Note that it is essential that the value of **calcCost** after the move is made must be the same as that of **calcChange** before it is made.

makeMove should make the given move on **conf**. **disposeMove** will be called to free any storage associated with the move. If there is no such storage that need be freed this parameter may be NULL.

If **loopFunc** is non-NULL then it will be called after each temperature point.

probsize is a measure of how large the problem is, and is used to determine some parameters. **maxmoves** is a rough estimate of how many moves are possible from any one configuration.

flags is a bitwise or of any of the following:

ADAPT_VERBOSE
: Print lots of messages.

ADAPT_PARANOID
: After every move, call **calcCost** and verify that the cost is what it should be. This will probably slow down the annealing quite a bit so it should not be used unless the cost function is being debugged. Actually **calcCost** will be called after every temperature point to verify that things are ok even without the **ADAPT_PARANOID** option.

If the **graphs** argument is non-NULL, it will be taken as a prefix for the names of some files that are created to hold statistics. Currently these files will contain the cost, the average cost, the temperature, and the high and low values of the "within window" (see the paper by Romeo for a description of the within count), all as a function of time. The graphs can be viewed by "cat *gfile** | xgraph".

The **quality** parameter is a fudge factor between 0 and 1 that should control the quality of the annealing, which should be inversely proportional to its speed. It's not clear what this should do. A value of 0.5 should yield good results.

adaptiveAnneal will return the final cost of the configuration after the annealing is completed.

SEE ALSO

"Probilistic Hill Climbing Algorithms: Properties and Applications", Fabio Romeo, U. C. Berkeley.

"Combinatorical Optimization, Simulated Annealing, and Fractals", Gregory Sorkin, U. C. Berkeley.

AUTHOR

Wayne A. Christopher

BUGS

The quality parameter is a hack and shouldn't be necessary.

Bibliography

[1] G. Adams. K. Roenner. T. Scheller. and P. Tzeng. Sogolar: Sea-of-gates optimized layout and routing system. EE 292H Report. Fall 1988.

[2] A. Aho. J. Hopcroft. and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley. 1974.

[3] S. B. Akers. *Routing.* chapter 5. pages 283–334. Prentice-Hall. 1972.

[4] M. Beardslee. J. Burns. A. Casotto. M. Igusa. F. Romeo. and A. Sangiovanni-Vincentelli. Mosaico: An integrated macro-cell layout system. In *MCNC International Workshop on Placement and Routing.* 1988.

[5] M. Beardslee, M. Igusa, A. Kramer. A. Sharma. G. Sorkin. and A. Sangiovanni-Vincentelli. ORCA: A sea-of-gates place and route system. In *MCNC International Workshop on Placement and Routing.* 1988.

[6] D. Braun. J. Burns. S. Devadas. H. Ma. K. Mayaram. F. Romeo. and A. Sangiovanni-Vincentelli. Chameleon, a new multi-layer channel router. In *Design Automation Conference Proceedings.* 1986.

[7] R. Brayton. E. Detjens. S. Krishna. T. Ma. P. McGeer. L. Pei. N. Phillips. R. Rudell. R. Segal, A. Wang. R. Yung. and A. Sangiovanni-Vincentelli. Multiple-level logic optimization system. In *ICCAD Digest of Tech. Papers.* 1986.

[8] J. B. Brinton. CHAS seeks title of global CAD system. *Electronics.* pages 100–102. February 1981.

[9] M. Burstein. Hierarchical channel router. In *Design Automation Conference Proceedings.* 1983.

[10] M. Burstein and R. Pelavin. Hierarchical wire routing. *IEEE Trans. on CAD of ICAS.* pages 223–234, 1983.

[11] W. Christopher. A. Burstein, T. Collins. and L. Layer. A placement program for sea of gates. EE 244 Report. Fall 1986.

[12] W. Dai. H. Chen. R. Dutta, M. Jackson. E. S. Kuh. M. Marek-Sadowska. M. Sato. D. Wang. and X. Xiong. BEAR: A new building-block layout system. In *ICCAD Digest of Tech. Papers.* pages 34–37, 1987.

[13] W. Dai and E. S. Kuh. Simultaneous floor planning and global routing for hierarchical building-block layout. *IEEE Trans. on CAD*. CAD-6(5):828–837. 1987.

[14] C. M. Davis. IBM System 370 bipolar gate-array microprocessor chip. In *Proc. IEEE Int. Conf. Circ. and Computers*. pages 669–673. October 1980.

[15] W. A. Dees and R. J. Smith. Performance of interconnection rip-up and reroute strategies. In *Design Automation Conference Proceedings*. pages 382–390. 1981.

[16] S. Devadas. *Techniques for optimization-based synthesis of digital systems*. PhD thesis. U. C. Berkeley. 1988. UCB/ERL Memo M88/54.

[17] W. E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Trans on Circuits and Systems*. CAS-26(4). April 1979.

[18] A. E. Dunlop and B. W. Kernighan. A procedure for layout of standard-cell VLSI circuits. *IEEE Trans on CAD of ICAS*. CAD-4(1):92–98. January 1985.

[19] B. Eschermann. Hierarchical placement for macrocells with simultaneous routing area allocation. Master's thesis. U. C. Berkeley, 1988. UCB/ERL Memo M88/49.

[20] C. M. Fiduccia and R. M. Matheyses. A linear time heuristic for improving network partitions. In *Design Automation Conference Proceedings*, 1982.

[21] A. El Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Trans on Circuits and Systems*. CAS-28(2). Febuary 1981.

[22] M. R. Garey and D. S. Johnson. *Computers and Intractability : a Guide to the Theory of NP-completeness*. W. H. Freeman. 1979.

[23] G. B. Goates. ABLE: A LISP-based layout modeling language with user-definable procedural models for storage/logic array design. Master's thesis. University of Utah. December 1980.

[24] D. Harrison. P. Moore. R. Spickelmier. and R. Newton. Data management and graphics editing in the berkeley design environment. In *ICCAD Digest of Tech. Papers*. 1986.

[25] W. R. Heller. W. F. Mikhail. and W. E. Donath. Prediction of wiring space requirements for lsi. *Journal of Design Automation and Fault-Tolerant Computing*. pages 117–144. 1978.

[26] D. W. Hightower. A solution to line routing problems on the continuous plane. In *Design Automation Conference Proceedings*. 1969.

[27] H. P. Holzapfel. K. H. Horninger. and P. Michel. Design and application of a 20[illegible] gate array. *IEEE Trans. and Ind. Electr*, 1E-33(4), Nov 1986.

[28] Y. Horiba. A bipolar 2500-gate subnanosecond masterslice LSI. In *Digest of Technical Papers, IEEE Int. Solid State Circuits Conf.*. pages 228–229, New York. New York. February 1981.

[29] C. P. Hsu. R. A. Perry. S. C. Evans. J. Tang. and J. Y. Liu. Automatic layout of channelless gate array. In *Proceedings of the Custom Integrated Circuits Conference.* pages 281–284. 1986.

[30] C. P. Hsu. R. A. Perry. S. C. Evans. J. Tang. and J. Y. Liu. An effective hierarchical approach to high complexity circuit layout. In *Proceedings of the Custom Integrated Circuits Conference.* 1987.

[31] M. Iacopona. D. Vail. S. Bierly. and A. Ignatowski. A hierarchical gate array architecture and design methodology. In *Design Automation Conference Proceedings.* pages 439–442. 1985.

[32] D. Johansen. Bristle Blocks: A silicon compiler. In *Proc. 16th Design Automation Conference.* pages 310–313. June 1979.

[33] J. W. Jones. Array logic macros. *IBM J. Res. Develop..* pages 98–109. March 1975.

[34] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal.* 47(2):991–308. 1970.

[35] S. Kirkpatrick. C. D. Gelatt Jr. and M. P. Vecchi. Optimization by simulated annealing. *Science.* 220:671–680. 1983.

[36] T. Kozawa. H. Terai. T. Ishii. M. Hayase. C. Miura. Y. Ogawa. K. Kishida. N. Yamada. and Y. Ohno. Automatic placement algorithms for high packing density VLSI. In *Design Automation Conference Proceedings*, pages 175–181. 1893.

[37] H. Kubosawa and Some Other People. Layout approaches to high-density channelless masterslice. In *Proceedings of the Custom Integrated Circuits Conference.* 1987.

[38] E. S. Kuh and M. Marek-Sadowska. *Layout Design and Verification.* chapter Global Routing. pages 169–198. North-Holland. 1986.

[39] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Trans on Computers.* C-20(12):1469–1479. 1971.

[40] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. In *Design Automation Conference Proceedings.* 1979.

[41] L. Layer. An analysis of sea of gates template and cell library design issues. Master's thesis, U. C. Berkeley, 1988. UCB/ERL Memo M88/8.

[42] B. Lee. Experiments in hierarchical routing of general areas. Master's thesis. U. C. Berkeley, 1989. to be published.

[43] C. Lee. An algorithm for path connection and its applications. *IRE Trans. Electronic Computers.* pages 346–365, 1961.

[44] J. T. Li and M. Marek-Sadowska. Global routing for gate arrays. *IEEE Trans. on CAD of ICAS.* pages 298–308. 1984.

[45] K. Y. Liao. C. P. Hsu. and M. R. Chin. Triple-level metal gate array using channelless architecture. In *Proceedings of the Custom Integrated Circuits Conference*. 1987.

[46] R. Linsker. An iterative-improvement penalty-function-driven wire routing system. *IBM Journal of Research and Development*. 28(5):613–624. 1984.

[47] S. Mallela and L. K. Grover. Clustering based simulated annealing for standard cell placement. In *Design Automation Conference Proceedings*. 1988.

[48] M. Marek-Sadowska. Private communication.

[49] Microelectronics Design Division. Institute of Technology Development. P. O. Drawer EE. Mississippi State University. MS 39762. *Scalable CMOS (SCMOS) Standard Cell Family*. USC-ISI (Contract No. MDA-903-86-C-0016).

[50] E. F. Moore. The shortest path through a maze. *The Annals of the Computation Laboratory of Harvard University*. 3(2):285–292. 1959.

[51] B. T. Murphy. A CMOS 32b single-chip microprocessor. In *Digest of Technical Papers. IEEE Int. Solid State Circuits Conf.*. pages 230–231. New York. New York. February 1981.

[52] R. Nair. A simple yet effective technique for global wiring. *IEEE Trans on CAD*. CAD-6(2):165–172. 1987.

[53] R. Nair. S. J. Hong. S. Lies. and R. Villani. Global wiring on a wire routing machine. In *Design Automation Conference Proceedings*. 1982.

[54] Y. Nishizaki. A cell library implementation on the siemens sea of gates template. EE 290H Report. Fall 1987.

[55] G. Odawara and Some Other People. Partitioning and placement technique for CMOS gate arrays. *IEEE Trans CAD*. pages 355–363. May 1987.

[56] C. J. Poirier. EXCELLERATOR: Automatic leaf cell layout agent. In *ICCAD Digest of Tech. Papers*. pages 176–179. 1987.

[57] F. Romeo and A. Sangiovanni-Vincentelli. *Probabilistic Hill-Climbing Applications: Properties and Applications*. Computer Sciences Press. Chapel Hill. N. C.. 1985.

[58] K. Sawada. T. Sakurai. K. Nogami. T. Iizuka. Y. Uchino. Y. Tanaka. T. Kobayashi. K. Kawagai. E. Ban. Y. Shiotari. Y. Itabashi. and S. Kohyami. A 72K CMOS channelless gate array with embedded 1Mbit dynamic RAM. In *Proceedings of the Custom Integrated Circuits Conference*. 1988.

[59] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE Journal of Solid-state Circuits*. pages 510–522. April 1985.

[60] C. Sechen and A. Sangiovanni-Vincentelli. TimberWolf3.2: A new standard cell placement. global routing package. In *Design Automation Conference Proceedings*. pages 432–439. 1986.

[61] H. Shin. *Two Dimensional Routing and Compaction in Computer-Aided Design of Integrated Circuits.* PhD thesis. U. C. Berkeley. 1987. UCB/ERL Memo M87/92.

[62] H. Shin and A. Sangiovanni-Vincentelli. Mighty: A rip-up and reroute detailed router. In *ICCAD Digest of Tech. Papers.* 1986.

[63] H. Shin and A. Sangiovanni-Vincentelli. A detailed router based on incremental routing modifications: Mighty. *IEEE Trans on CAD.* CAD-6(6):942–955. 1987.

[64] G. Sorkin. Combinatorial optimization. simulated annealing. and fractals. Research Report RC 13674 (61253). IBM. 1988.

[65] P. Suaris and G. Kedem. Quadrisection: A new approach to standard cell layout. In *ICCAD Digest of Tech. Papers.* pages 474–477. 1987.

[66] Y. Suehiro. D. Miura. M. Naitoh. S. Tsutsumi. and T. Shirato. A 120k-gate usable cmos sea of gates packing 1.34m transistors. In *Proceedings of the Custom Integrated Circuits Conference.* 1988.

[67] J. D. Trotter. R. Hiltpold. J. Kelly. R. Elling. and M. Landrum. *Scalable CMOS Design Rules for Fabrication through the MOSIS Silicon Foundries.* University of Southern California. P.O. Box 77967, University Park. Los Angeles. CA 90007.

[68] R. Tsay. E. S. Kuh. and C. P. Hsu. PROUD: A fast sea-of-gates placement algorithm. In *Design Automation Conference Proceedings.* 1988.

[69] P-S. Tzeng and C. H. Séquin. Codar: A congestion-directed general area router. In *ICCAD Digest of Tech. Papers.* 1988.

[70] M. Watanabe. CAD tools for designing VLSI in Japan. In *Digest of Technical Papers. IEEE Int. Solid State Circuits Conf..* pages 242–243, Philadelphia. Penn.. February 1979.