Cost calculation for incremental hardware synthesis

van Engelshoven, R.J.; Born, van den, R.

Published: 01/01/1988

Citation for published version (APA):
Cost Calculation for Incremental Hardware Synthesis

by
R.J. van Engelshoven
and
R. van den Born

EUT Report 88-E-202
ISBN 90-6144-202-8
June 1988
COST CALCULATION FOR INCREMENTAL HARDWARE SYNTHESIS

by

R.J. van Engelshoven
and
R. van den Born

EUT Report 88-E-202
ISBN 90-6144-202-8

Eindhoven
June 1988
COOPERATIVE DEVELOPMENT OF AN INTEGRATED, HIERARCHICAL AND MULTIVIEW VLSI DESIGN SYSTEM WITH DISTRIBUTED MANAGEMENT ON WORKSTATIONS.

(Multiview VLSI-design System ICD)

code: 991

DELIBERABLE

Report on activity: S.1.B.

Abstract: The problem addressed in this report is the incremental estimation of layout area for logic circuits. The estimates are to be used during the automated synthesis of hardware structures. These estimates are used to compare circuits on their efficiency. The absolute accuracy of the estimate, with regard to the real area occupation of the circuit, is less important than the relative accuracy with respect to the other circuits in the comparison. Two estimation methods are investigated: one based on linear placement and the other based on a probabilistic estimation scheme.

The former method estimates area by placing the components in a linear sequence with the interconnecting wires situated along this sequence. The area is then determined by the rectangle enclosing both the components and the wires.

The linear placements are calculated incrementally: i.e. when a component is added to the circuit (part of) the old placement is reused.

The second method is useful for very fast estimates on large circuits that have a stable average wire length.

deliverable code: WP 5, task: 5.1, activity 5.1.B

date: 06-04-1988

partner: Eindhoven University of Technology

authors: R.J. van Engelshoven, R. van den Born

This report was accepted as a M.Sc. Thesis of R.J. van Engelshoven by Prof. Dr. - Ing. J.A.G. Jess, Automatic System Design Group, Faculty of Electrical Engineering, Eindhoven University of Technology. The work was supervised by Drs. R. van den Born.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Engelshoven, R.J. van

Cost calculation for incremental hardware synthesis / by R.J. van Engelshoven and R. van den Born. - Eindhoven: University of Technology. - Fig., tab. - (Eindhoven University of Technology research reports / Faculty of Electrical Engineering, ISSN 0167-9708; 88-E-202)

Met lit. opp., reg.
ISBN 90-6144-202-B
S150 664.3 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.
CONTENTS

1. INTRODUCTION
   1.1 Overview of the report

2. THE SILICON COMPILER
   2.1 SYSTEM ARCHITECTURE
   2.2 DYNAMIC PROGRAMMING
   2.3 STATE GENERATION
   2.4 HARDWARE GENERATION
   2.5 CIRCUIT SYNTHESIS, AN EXAMPLE
   2.6 PROBLEM FORMULATION

3. RELATED WORK
   3.1 A SURVEY OF SOME AREA ESTIMATION STUDIES
      3.1.1 WIRING SPACE ESTIMATION FOR GATE ARRAYS
      3.1.2 WIRING SPACE ESTIMATION OF CUSTOM LAYOUTS
      3.1.3 GENERAL AREA ESTIMATION
      3.1.4 CONCLUSIONS
   3.2 LINEAR PLACEMENT METHODS
      3.2.1 GRAPH THEORETIC APPROACH
      3.2.2 CLUSTERING ALGORITHMS
      3.2.3 CONCLUSIONS

4. MODEL PROPOSAL
   4.1 ASSUMPTIONS
   4.2 BIT SLICES AND STANDARD CELL METHODOLOGY

5. MODEL DESCRIPTION
   5.1 PROGRAM ARCHITECTURE
   5.2 CIRCUIT ALTERATIONS
      5.2.1 CIRCUIT EXTENSION
      5.2.2 CIRCUIT REMOVALS
      5.2.3 MODULE SUBSTITUTIONS
   5.3 PLACEMENT AND RETRACTION
      5.3.1 BUILDING A LINEAR SEQUENCE
      5.3.2 INITIAL SEED SELECTION
      5.3.3 GLOBAL NETS
      5.3.4 UPDATE PLACEMENT STRATEGIES
   5.4 COST CALCULATION
   5.5 PROGRAM COMPLEXITY
   5.6 CRITICAL PATH ANALYSIS

6. PROBABILISTIC MODEL FOR AREA ESTIMATION
   6.1 MODEL DESCRIPTION
   6.2 AVERAGE WIRE LENGTH
   6.3 REMARKS

7. PROGRAM IMPLEMENTATION
   7.1 ASSUMPTIONS
   7.2 INTERFACES
   7.3 INTERNAL DATA STRUCTURES
   7.4 INTERNAL VECTORS
   7.5 INPUT & OUTPUT
   7.6 FACILITIES FOR INCREMENTAL USE
   7.7 EXPERIMENTAL RESULTS
8. SUGGESTIONS FOR FURTHER RESEARCH ........................................ 45
9. CONCLUSIONS ........................................................................... 46
    REFERENCES ............................................................................ 47
    APPENDIX A ........................................................................... 51
    APPENDIX B ........................................................................... 57
    APPENDIX C ........................................................................... 59
    APPENDIX D ........................................................................... 60
    APPENDIX E ........................................................................... 61
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>hardware synthesis system</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of a demand graph</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>The Node-Set lattice of previous demand graph</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Two realizations for node 7 and 8</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Part of implementation graph</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Examples of two possible resulting circuits</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Spanning nets and fill-in nets</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Required area for one slice</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>A bit-slice organised data path</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Program architecture</td>
<td>20</td>
</tr>
<tr>
<td>5.2</td>
<td>List of circuit extensions</td>
<td>21</td>
</tr>
<tr>
<td>5.3</td>
<td>List of circuit removals</td>
<td>22</td>
</tr>
<tr>
<td>5.4</td>
<td>Set updating</td>
<td>24</td>
</tr>
<tr>
<td>5.5</td>
<td>Different nets</td>
<td>25</td>
</tr>
<tr>
<td>5.6</td>
<td>Linear placement of well structured circuit</td>
<td>26</td>
</tr>
<tr>
<td>5.7</td>
<td>Selecting circuit partitions</td>
<td>27</td>
</tr>
<tr>
<td>5.8</td>
<td>Local and global nets</td>
<td>28</td>
</tr>
<tr>
<td>5.9</td>
<td>Model with track profile</td>
<td>32</td>
</tr>
<tr>
<td>5.10</td>
<td>Run time of cost calculation program</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>The row model</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Density at point X</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Syntax of both external network vectors</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>Syntax of net duo list</td>
<td>41</td>
</tr>
<tr>
<td>7.3</td>
<td>Syntax of input sequence and output list</td>
<td>43</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 1. Distribution of both seed selection groups with respect to final net area requirement. .......................... 57
TABLE 2. Performance results for two different seed selection rules. ................................................................. 58
TABLE 3. Impact of net size on the run time and average size of ACTIVE. ......................................................... 60
TABLE 4. Distribution of netweights for eleven benchmark circuits. ................................................................. 60
TABLE 5. Experimental and theoretical wire length distribution. ................................................................. 61
1. INTRODUCTION

This report describes the selection and implementation of an area, time, and power estimation method for VLSI circuits described at the structural level. It provides an accurate and efficient method for estimating area consumption of circuits described at the structural level.

This report describes the results of a graduation project at the Automatic System Design group of the department of Electrical Engineering of the Eindhoven University of Technology. Research in this group is aimed at the development of CAD tools for VLSI design.

Advances in technology allow increasingly complex circuits. The design cost of such circuits and the probability of design errors rise accordingly. Therefore there is a need for tools that automate circuit design. One field of research in this respect is silicon compilation, the translation of a behavioural or functional specification of a circuit into a layout. It promises a higher level of abstraction to reduce the design complexity and correctness by construction. The first developments in this field were published in the early eighties. Today many technical universities and industries put effort in developing these tools.

A early step in silicon compilation is the hardware generation. This comprises the synthesis of a structural design from the functional specification. One method still under investigation is based on dynamic programming. It generates simultaneously sets of different hardware implementations for a single specification. During this process implementations are compared and the better ones selected.

The selections are made using a cost function that may take area, time, and power consumption into account. To make a proper choice these parameters must be estimated as accurate as possible. On the other hand the estimations have to be made very often, inducing the use of a fast method. This report describes the selection and implementation of two estimation methods that are sufficiently accurate and efficient. Although selected with the dynamic programming method in mind they can be applied in any field where parameter estimates for circuits described at the structural level are needed.

1.1 Overview of the report

Chapter 2 gives an overview of the silicon compiler project. The place of the estimator within the tool is considered and the conditions it has to meet are derived. A survey of previous research on the topic of area estimation is presented in chapter 3. The survey includes also several linear ordering algorithms anticipating the estimation scheme presented in the following chapters.

The first algorithm uses a linear ordering method to estimate area. Chapter 4 contains a description of the model used by this algorithm. The algorithm itself is described in chapter 5. The second method is more theoretical using empirically derived equations for average wire length and wire length distribution. It is faster but less accurate than the first. This method is described in chapter 6. In chapter 7 details of the implementation of the first method are given. It also describes the interface of the tool and the hardware generator.

Finally chapters 8 and 9 contain suggestions and conclusions concerning the methods used.

1. This research is sponsored by the European Community under contract ESPRIT991.
Appendix B gives the results of some experiments performed to test the quality of the placement as generated by the first algorithm.
2. THE SILICON COMPILER

Before proceeding with the hardware generator and the application of an area estimator I will first in short describe the silicon compiler already mentioned in the introduction. A more elaborate definition of the system can be found in [Stok86] and [Born87].

I will pay special attention to the hardware generator based on dynamic programming as this tool evaluates its results using a cost function. This function depends on estimates of critical path delay, power consumption and area occupation. To illustrate the hardware synthesis process some implementation steps resulting from the presented example are worked out in detail.

2.1 SYSTEM ARCHITECTURE

In figure 2.1 a global scheme of the system is presented. The system is partitioned in several intermediate results and tools. The tools (shown in ellipses) convert the intermediate results (shown in boxes) to each other.

![Diagram](image)

Figure 2.1. hardware synthesis system.

A high level description of a system is used as input for the silicon compiler. This description is a behavioral description. This description deals with the functions to be implemented and requirements concerning power, reliability, area, pin out, timing, technology etc. to be fulfilled. This high level description can be given in a language like Pascal, C or Lisp.

The algorithmic description of the system is syntactically analyzed by the parser and is converted
into an abstract syntax-tree. This can be done using conventional compiler techniques. The demand-graph-constructor transforms the syntax tree in a demand graph (figure 2.2). The demand graph represents both data flow and control flow of the system described in the input language. Nodes represent the operations on the data. The arrows indicate the direction in which the data flows.

In figure 2.2 variables $v_1$, $v_2$, $v_4$, $v_6$ and $v_{10}$ are externally provided to the system. Through the get nodes, performing an equivalent of a read, the variables become available in the system. An operation may be performed if all required input values are present. Thus the operation represented by node 8 has to wait until the output from node 7 is ready. In the end outputs are returned by the put nodes.

![Figure 2.2. Example of a demand graph](image)

The optimizer converts a demand graph to a functionally equivalent demand graph. These conversions are done because they will result in a more efficient implementation of the algorithm. Most optimizations are similar to those used in optimizing compilers.

### 2.2 DYNAMIC PROGRAMMING

Dynamic programming is an optimization strategy for a class of problems called multi-stage decision processes. The following features can be identified for those problems:

a. The processes are characterized at any stage by a small set of parameters, the state variables.

b. At each stage of these processes there is a choice of a number of decisions.

c. The effect of a decision is a transformation of the state variables.

d. The history of the system is of no importance in determining future actions.
The purpose of the process is to maximize some function of the state variables.

The dynamic programming approach generates a limited set of solutions for these problems among which the optimal solution is present. An exhaustive description of dynamic programming is given in [Bellman62].

The hardware synthesis problem can be described as an N-step deterministic decision problem. Deterministic here means that the result of a decision is uniquely determined by the decision. In fact it is a multi-step allocation problem.

In the solution of such problems by dynamic programming, we rely on the principle of optimality:

An optimal policy has the property that whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

A policy is any rule for making decisions which yield an allowable sequence of decisions. This principle of optimality guarantees that an optimal policy is found.

2.3 STATE GENERATION

In this section the way states are generated is described.

Generating an efficient implementation of a demand graph requires the following interdependent tasks to be performed:

- A network of registers, operators, multiplexers and controllers has to be constructed.
- Instructions (nodes) are to be mapped to operators and machine cycles.
- Edges are to be mapped to busses and, possibly to registers if the adjacent instructions are assigned to different machine cycles.

During the hardware generation the demand graph nodes are implemented one by one by a dynamic programming approach. In the demand graph the nodes represent the operations on the data and the arcs represent the data dependencies. A node in the demand graph is ready for implementation if all variables needed by this node (incoming arcs) are available. So a node is called free if all the nodes these variables depend on have been implemented.

Each time a demand graph node is implemented a new state is created. A state is characterized by a set of implemented demand graph nodes. In this way a state graph or nodeset–lattice is generated (see fig. 2.3). This graph represents the process flow of the hardware generation. In the figure the numbers in the graph vertices correspond to the demand graph node numbers of figure 2.2 and represent the set of free nodes of the corresponding state. The graph is strictly leveled. That is the level of a state is determined by the number of implemented nodes for that state.

A new state can be generated from a previous state by implementing a node or a collection of nodes from that state's free node set. If n free nodes are associated with a state, in the worst case n new states may be generated from that state. These new states again have sets of free nodes, each of which again may result in a new state. It is clear that by this feature an excessive growth of the state graph can occur.
2.4 HARDWARE GENERATION

The synthesis of hardware for the demand graph is done by going over the state-graph level by level. For each state a partial implementation is maintained. While generating a new level free nodes are implemented on top of the partial implementations of the old level. A node may be implemented in different ways, each giving rise to a new state on the next level. As a result of these successive implementations of nodes an implementation graph is created. This graph is closely related to the state-graph. But as a consequence of the various implementations possible for a node in the state graph, the implementation graph may have several nodes where the state graph has only one.

Implementing free nodes on top of a partial implementation can be done either by using an existing piece of hardware or instantiating a new operator. Furthermore a choice has to be made between allocating the node in the current machine cycle or in the next. The latter would be necessary if no appropriate hardware is free and/or if the length of the clock cycle of the circuit would
grow too large.

At the current stage of the project all four possible implementations are generated and the selection is left to the dynamic programming process. This approach guarantees for a simultaneous optimization of delay and area requirement.

Two main mechanisms, the cost function and the concept of comparability of implementations, control the generation process. They delimit the number of states that coexist at one level. The cost function is a function of the process state variables which represent quantities as area, time and power consumption of the partial implementations. The cost function is employed each time a new state is generated. It calculates the costs for the implemented hardware which is stored as a property of the state. It is on the basis of these costs that decisions are made with which implementation to pursue and which to eliminate.

The notion of two implementations being comparable was introduced to prevent initially expensive but interesting implementations to perish too soon. On the other hand this mechanism may not result in maintaining too many implementations as this may result in a wide graph and excessive computation time. Several heuristics may be applied to meet this problem. Currently two implementations are defined comparable if:

- they implement the same demand graph nodes
- they have the same number of major operators

where major operators are the ones that are considered important because of their size, speed or power consumption.

This definition of comparable states implies that comparable states can only exist at the same level. Consequently, only states on the same level need to be checked on comparability. In fact only the states on one level are maintained in the dynamic programming approach. If two states are found to be comparable the most expensive state and according subtree is discarded from the implementation graph.

2.5 CIRCUIT SYNTHESIS, AN EXAMPLE

To illustrate the hardware synthesis process some implementation steps are worked out for the realization of the system described by the demand graph of figure 2.2. The state graph of figure 2.3 visualizes this process. The situation where all input variables \( v_1 \), \( v_2 \), \( v_4 \), \( v_6 \) and \( v_{10} \) are available but no nodes have been implemented yet corresponds with the state at the top of the state-lattice, characterized by free node 7. This means that at this state only node 7, being the addition of \( v_1 \) and \( v_2 \), can be performed and thus implemented. As no components are yet present in the circuit the first circuit element will be an instantiation of an adder.

Having implemented node 7 of the demand graph the process has made a transition to the next state denoted by free nodes 8 and 9, representing resp. a subtraction and a multiplication. Both nodes can be implemented since their input values are available. Implementing node 9 passes the process to the right next state, while a transition to the left next state is made by implementing node 8. This state indicated by free nodes 9, 10 and 12 may represent one or more implementation states, for the second addition/subtraction operation can be realized in several manners. A first implementation (figure 2.4 a) is to allocate a second instantiation of an adder. An alternative is to use the already present adder and to assign the subtraction operation to a next clock cycle. This implies the use of four additional registers to store intermediate values of variables, and two multiplexers (figure 2.4 b).

Although the construction of the latter circuit may be more expensive than the former this early use of additional hardware may prove economical in a future phase of the process. Therefore these
two implementations are allowed to co-exist, they are not comparable.

By implementing free nodes of states on level 2 transitions are made to level 3 states. Expansion of the partial circuits of level 2 with resp. implementation of node 9 and 8 result in the level 3 state denoted by free nodes 10, 11 and 12. Behind this state 3 implementations now exist, two of which are comparable. The implementation graph for this part of the state graph may look like figure 2.5. One of the implementation paths resulting in a comparable state is discarded and two different implementations remain.

It should be pointed out that even more implementations are possible for the described nodes, depending on the available library cells, the flexibility of the hardware generator and the constraints imposed on the synthesis process. Finally figure 2.6 shows two possible circuits for the demand graph.

The left most circuit performs all operations in one machine cycle, needs 4 adders, 2 multipliers and 2 and/or operators. The second implementation requires one multiplier, 2 adders, 2 and/or operators, 5 registers, 6 multiplexers and needs two machine cycles to perform the same function.
2.6 PROBLEM FORMULATION

After the previous investigating sections it is time to identify the problem with respect to cost calculation and formulate the research issues for our problem.

As was made clear in the previous section, the hardware generator creates a set of different implementations for the same algorithm in which the optimal solution is included. A major disadvantage of this method may be the exponential growth of the process graph if no good restriction tool can be applied. This restriction tool can be thought of as some quantity supplied by a cost estimating mechanism to evaluate comparable implementations. As a result of an evaluation the cheapest implementation is saved, the others are removed from the implementation graph. Such a procedure provides the synthesizer with cost estimates at the early stages of the design process, thereby avoiding a waste of resources and time in further pursuing the synthesis of a design which may not satisfy some early imposed constraints.

One important variable in such a cost function, although not the only one, is an area estimate. In the following the area estimation will be considered first. Relatively simple tools can provide estimates on delay and power consumption.

The routine for area estimation is very intensively called, so it has to provide quick cost estimates for a new implementation. The routine is called each time a new demand graph node is implemented. In general this means that one or a few modules have been added to the already existing implementation of a previous state. For this previous state the cost was already calculated. So, a considerable gain in efficiency can be achieved when the cost calculation is incrementally adjusted for new states.

For reasons of time efficiency area estimates should be calculated without actually doing the layout and routing. In many cases the area estimating unit is based on a not existing or another layout methodology than the one that is finally used in the design system. However the final layout methodology has a great impact on the final area requirement. This sounds like a paradox. In our case we can make use of the fact that we have to decide between different implementations of the same function. This means that the absolute area requirement is less important than the difference in costs for those alternative implementations. This is a very important assertion. Depending on the kind of circuit that is synthesized the most optimal layout methodology may be
used without conflicting with previous steps in the design process.

To summarize, the cost estimating unit should have the following features:

- very quick
- incremental
- model may be independent of the actually used layout methodology
- relative accuracy is required
3. RELATED WORK

This chapter comprises a summary of earlier research work done by others on area estimation. Further a summary of the most important linear ordering algorithms useful in the yet to describe heuristic area estimation schemes is included. This survey may help to get a better appreciation of the addressed problem and serve as a starting point for possible further research.

3.1 A SURVEY OF SOME AREA ESTIMATION STUDIES

There are mainly two approaches in the research of estimating layout area for electrical circuits. A theoretical approach, which concentrates on wirability analysis and wiring space estimation. Secondly an experimental, which aims at developing area estimators for some specific layout systems based on previous experience with these systems. In the next sections I discuss a number of articles on the subject found in the available literature.

The articles discussing wiring space estimation for gate arrays and for custom layouts all apply a theoretical approach. Also the method by Kurdahi is of a theoretical nature. The work by Ueda is strongly experimental oriented. All methods aim at estimating the final implementation area needed to implement a circuit.

3.1.1 WIRING SPACE ESTIMATION FOR GATE ARRAYS

Two important works can be mentioned here. Heller from IBM [Heller77] has proposed a stochastic model for the prediction of the wiring on a one dimensional (1-D) placement of cells. Given the average wire length, the number of devices and the average number of connections per device, the model predicts the probability of successfully routing the placement within the allocated space.

The problem of predicting the wire length distribution in regions of homogeneous logic was studied by Feuer [Feuer82]. The main conclusion of this work is: if the partitioning of a logic graph exhibits Rent's rule, then the wire length distribution is expected to be of the form \( q(r) = r^{-\beta} \) where \( \beta \) is Rent's exponent and \( r \) is the wire length. Rent's rule is an empirical relationship for predicting the number of external connections from a given number of components in well-partitioned computer logic. The relationship was originally formulated as \( l = bC^{p} \) where

- \( C \) is the number of components on a package
- \( b \) is the average number of connections per component, and
- \( p \) is a small positive exponent, originally fixed at 1/2.

For the sake of completeness it should be mentioned that [Yehc82] and [Sastry85] also have published on the problem but I was unable to obtain any of their publications.

3.1.2 WIRING SPACE ESTIMATION OF CUSTOM LAYOUTS

The above works assumed that designs are laid out in the gate array design style. The problem of wiring space estimation of custom logic was addressed by Syed [Syed81]. He assumes that a placement of arbitrarily-sized rectangular blocks is given, along with their interconnections. He constructs a channel graph for the placement, where edges are the routing channels between the blocks and vertices are the intersections of these channels. The widths of the channels (edges of the graph) are then estimated using a stochastic model for wiring in which pins are assumed to be
generated along a channel with a Poisson distribution, and wire lengths are exponentially distributed. Once this is done, the initial placement of the blocks is modified so as to accommodate the channel width estimates with minimal total displacement of the blocks.

Routing is performed in two phases. First topological routes are assigned to wires. In the second phase, tracks are assigned to wire segments. During this phase, the placement may be further modified if more space than predicted is needed in some channels. This process of track assignment and placement modification is repeated iteratively until complete routing is achieved. Syed's model concentrates on estimating the area needed for global routing between the major blocks of the chip and does not deal with estimating the local wiring area within blocks and hence, the area of blocks themselves.

Another approach to custom layout area estimation is by Ngai [Ngai83]. His model consists of a channel, a number of tracks allocated to it and a set of nets of three types: right and left nets entering the channel from right and left, respectively, and center nets which are born and die inside the channel. The problem is to estimate the routability of the channel over all possible pin permutations.

The channel wiring is modeled as a Markovian stochastic process with state changes occurring at net terminals (pins). A state at a pin is defined as a quadruple of random variables representing the number of nets of different types up to that pin, the density at that pin being a simple function of these variables. The conditional transition probabilities are found and a recurrence relation is used to find the state occupancy probabilities. From these the distribution of the density function is determined and the routability estimated. Since this model predicts routability over all possible pin permutations and not the subset corresponding to 'good' placements, the predicted routability figures may deviate from the actual ones.

3.1.3 GENERAL AREA ESTIMATION

In the field of general area estimation the work of Ueda et al. [Ueda85] can be noted. It is a more experimental approach to the area estimation problem. The authors describe a layout system, ALPHA, which uses the standard cell design style and in which an area estimator, CHAMP, is implemented. During the floor planning process, CHAMP estimates the areas of the different standard cell blocks by using empirical formulas obtained by running numerous layout experiments on several designs. The area estimation figures presented for some chips are within 10% of the actual area. The estimation formulas are, however, empirical in nature and no theoretical backing is provided. Hence it is doubtful whether such formulas are applicable in another system.

Kurdahi and Parker [Kurdahi85] have proposed a probabilistic model for standard cell area estimation. A basic assumption made for the model is the 1-D ordering and following folding placement technique which enables to better analyse the placement problem and therefore yields more confident area estimates. Another reason for making this assumption is that the 1-D placement and folding technique is a technique that is reported to produce very good results. Using the number of wires, the total length of all cells and the average wire length an estimate for track requirement is calculated for a 1-D placement of all cells on a single row.

A method is presented to apply the single row results to a folded row placement which delivers estimates for the total track and feedthrough requirements. Hence, an area estimate for a standard cell layout chip can be made. The obtained theoretical results for tracks and feedthroughs are verified on correctness and accuracy by comparing them with simulated results that do not incorporate any approximation methods. The accuracy of the approximated formulas is found to be within 4%.

A real data validation is done by comparing the model predictions to actual values of 'real' world cases. Six designs which were automatically layouted by the MP2D layout system [Feller78] are evaluated and area estimates were found to be within 10% of the actual layouts. Although the
developed model uses some simple wire and wire length distributions and only two terminal nets are considered, the results are promising.

3.1.4 CONCLUSIONS

A major drawback for using the first two methods in an area estimating scheme is the assumed infinite chip size. Due to that, it may incorrectly model the behaviour of a real 'finite' chip near the edges. Another disadvantage of Heller's approach is the fact that the model assumes a Poisson distribution for the number of wires originating at a given port, a generalisation certainly not applicable to every kind of circuit.

The relation for the wire length distribution derived by Feuer requires a good partitioning of the large (infinite) network. This is another reason for rejecting this approach for our problem. Our model will have to cope with circuits of arbitrary size. Because the hardware will be assembled from a great number of modules, partitioning of these modules will cost an unfavourable extra amount of processing time.

Syed's model assumes a placement, something we would like to avoid for time efficiency and generality of the model. Further an iterative placement-routing procedure is applied which will certainly slow down the system.

The other method developed for custom layouts predicts routability for random placed modules rather than for well placed logic which will yield pessimistic estimates. The applied technique of considering all pin permutations makes the method useless for large circuits.

The manner in which the formulas used in CHAMP were derived from actual layouts done by the system ALPHA suggests that these relations will only be useful for systems using the same layout methodology. However, these relations contain a great generality as Rent's rule can easily be derived from them. A reason for rejecting this estimating scheme is the fact that these empirical relations do not yield any relative accuracy in cases of minor differences between the considered logic.

Finally the probabilistic modeling by Kurdahi has some appealing characteristics. The hardware is modeled into a linear placement, a very time efficient method. The employed calculus for track requirement is quick and very accurate results are reported. However the method is indirect. The average wire length is one of the input parameters for the estimation of the required area. I suppose that they get this number from the linear placement modeling. It might be more accurate to calculate the real density and wire requirement from this placement than gain some time efficiency but yielding less accurate estimates. This wire length estimate may also be calculated by theoretical derived formulas. Some authors have been involved in deriving expressions for average connection length. Their published results, some of which are discussed later, may also be employed.

3.2 LINEAR PLACEMENT METHODS

Anticipating on the description of the model used for the cost estimation I will first give a summary of the most interesting linear placement strategies I have found.
3.2.1 GRAPH THEORETIC APPROACH

In [Fuji85] an heuristic algorithm for one-dimensional gate assignment is proposed which is reported to give very good solutions. The original minimization problem is transformed into a restricted problem in which each connection is restricted to be between two gates. Accordingly a randomly obtained placement all multipoint nets i.e. nets with more than two terminals, are substituted by a sequence of 2-terminal nets. The thus obtained new network a weighted graph \( G=(V,E) \) is constructed. Next, a new heuristic is applied to the restricted problem. The new algorithm works on graph \( G \) in which the nodes \( \{b_i,b_j\} \) represent the left most resp. the right most boundary of the layout region. In a first step vertices are clustered to \( b_i \) and \( b_j \) forming the sets \( C_i \) and \( C_j \). In a second and third step an ordering for the vertices in \( C_i \) and in \( C_j \) is determined iteratively. Finally all vertices of \( C_i \) and \( C_j \) in \( G \) are deleted and two super nodes are introduced. Now step 4 are repeated until all vertices are ordered. The solution obtained by the algorithm is interpreted as a solution to the original problem. The time complexity of the algorithm is given as \( O(|V|.|E|\log |E|) \). The statement that the algorithm produces optimal solutions is dubious as a best solution is selected out of a great number of runs each with different initial placements. The algorithm can be modified to perform incremental placements.

A totally different approach towards finding a one dimensional gate ordering is presented in [Ohtsuki79]. The problem is transferred into a graph-theoretic one based on finding a minimal clique number augmentation in a graph. Therefore a connection graph is constructed based on the net list of the circuit. The task is now to find an interval graph which has the least clique number by adding a set of edges. Unfortunately, this problem turns out to be NP complete. To find near optimal solutions a number of minimal augmentations are calculated to construct interval graphs based on randomly generated vertex orderings. Ohtsuki proposes an algorithm that generates a minimal augmentation in \( O(|V|.|E|+|F|) \) time wherein \( F \) represents the set of added edges. Of the interval graphs with least clique number all dominant cliques can be listed and ordered in \( O(|V|+|E|) \) time to receive a gate ordering which is a semi-optimum. The proposed method is not suited for an incremental placement.

Recently C.K. Cheng [cheng86] has proposed a linear placement algorithm that minimizes the sum of the wire lengths. The method is based on the cut-tree concept of Gomory and Hu [Gomory61]. When the cut tree is a chain, the sequence of this chain is optimal in terms of the sum of wire lengths and the maximum track density is a linear placement problem [Adolphson73]. To make this method more suited for general graphs, decomposition algorithms are introduced. It is assumed that two modules are fixed on both ends of the line for external connection and that all nets are 2-pin nets. The max-flow min-cut method [Ford56] is used to find a graph partitioning which establishes an optimal order. The main drawback of the method is that it is not always possible to partition the graph or parts of the graph in a useful manner. In these cases a relaxation scheme is used. The published experiments show that the new algorithm indeed realizes shorter total wire lengths than results published earlier.

3.2.2 CLUSTERING ALGORITHMS

Schuler and Ulrich [Schuler72] have proposed a two-step method for generating linear cell orders. They applied the method of clustering to find placements with near optimal results with respect to wire length. A clustering value is calculated for all connected pairs of elements which is a function of their mutual connection strength and their total connectivity. Each time the pairs with the greatest clustering values are clustered, this finally delivers a clustering tree. This tree represents a linear ordering which may be optimized by rotating subclusters about vertices in the tree. The run time of both proposed algorithms is close to linear in the size of the network.
Two interesting algorithms for the linear ordering problem were published by Goto [Goto77]. The aim of both algorithms is to minimize the maximum number of interconnecting wires, i.e., the density of the routing channel, between circuit boards. The process of finding a solution can be modeled into a state-space graph \( G = (V, A) \). This state-space graph displays a close resemblance to the state graph derived from the demand graph. From an early state a succeeding state is generated by adding a new module to the placement of the preceding state.

The first algorithm (the E-algorithm) finds a solution with a cost within a factor \((1+E)\) from the minimum. This is possible by allowing the algorithm to proceed with a state that has \((1+E)\) times the cost of the cheapest alternative state. The cost of a state is defined as the minimal density needed to interconnect the elements characterizing the state. In this way the memory space \( M \) and the computation time \( T \) is reduced. For a smaller value of \( E \), the computation time \( T \) and memory requirement \( M \) tend to be larger than for a larger value of \( E \).

The second algorithm (the C-algorithm) always chooses the cheapest state among the states closest to the final state. Selection is done over all unselected components. From experiments the C-algorithm appears to have \( O(n^2) \) complexity.

In [Asano82] T. Asano presents two algorithms for gate placement of MOS one-dimensional arrays. Instead of searching for an optimal gate ordering, an optimal net ordering is searched for. The corresponding gate ordering is constructed according to the net ordering obtained. The first described algorithm finds an optimal solution, the second a near optimal. The optimal solution is found using a branch and bound method. Its complexity varies from \( O(n^2) \) to \( O(n!) \) in the worst case, where \( n \) is the number of nets. A method of node domination is used to restrict branching but still guarantees an optimal solution. The algorithm is described in great detail and a formal proof for the optimality of the approach is provided. The second algorithm uses a greedy method to find a near optimal solution. It augments a partial net ordering by choosing a best estimated net at each stage in \( O(n \log n) \) or \( O(n^2) \) time.

Another linear placement heuristic was developed by S. Kang [Kang83]. He has presented a new strategy for linear ordering. One notable difference of this strategy from the previous ones is that it starts the ordering process with the most lightly connected seed (first selection) whereas most methods start with the most heavily connected seed. After the seed selection the ordering objective is to minimize the connections between already selected modules and yet-to-be selected modules. It is equivalent to minimizing the number of nets crossing a cut plane (net-cuts) in a linear sequence, or to minimize the maximum number of net-cuts [Schweikert72]. The technique has been applied for standard cell, gate array and linear placement with good results [Kang83] [Reingold84] [Green86]. After minor modification the heuristic is suited for incremental use.

3.2.3 CONCLUSIONS

Apart from the clustering scheme of [Schuler72] all mentioned heuristics produce reasonable results. Asano’s algorithm even delivers optimal results. Recalling the need for relative accuracy of the cost estimates, all kind of heuristics employing some kind of random generation are unfavourable. Methods working on the whole network of the circuit at a time usually do not allow for any incrementality.

A general remark concerning the graph theoretic approaches is the excessive amount of overhead they tend to bring about, like graph modeling, adjusting and analyzing. If we only restrict to polynomial time heuristics only three of the discussed algorithms are suited.

The most striking disadvantage of Asano’s approximation algorithm is that net selection is done over the collection of all unselected nets. Together with the fact that the number of nets in a circuit generally is larger than the number of components, this algorithm will be considerably slower than Kang’s heuristic. Moreover, all the linear sequences I generated with the Asano net ordering
strategy proved to be worse than those obtained with Kang’s strategy. Introducing a third group of all possible candidates for selection will not work either. Frequently, a net not connected to already placed modules must be selected to obtain a reasonable placement.

But the main disadvantage of selecting nets is the inevitable accompanying component selection. If a net is selected, the unplaced modules connected to this net will be considered in a next selection step. These modules are the first to join the partial placement. This however does not resemble most placements. In a linear placement there are often nets that span a lot of components without actually having a terminal on them. These spanned components are connected to fill-in nets. Figure 3.1 shows a placement with spanning and fill-in nets.

![Figure 3.1. Spanning nets and fill-in nets.](image)

The use of an intermediate collection of modules among which a best suited next module for the placement is selected is a clever enhancement of the C-algorithm by Goto. I expect the latter to have smaller complexity because of the considerable shorter selection effort needed. For all reasons described I chose Kang’s linear ordering algorithm to do the placement.

It should be pointed out here that the previous described heuristics all aim at obtaining near optimal results. In a recent publication [Deo87] it is shown that there are however families of problem instances for which the ratio of tracks required by these heuristics to the optimal value is unbounded. This result holds for any heuristic algorithm. In the same article it was shown that, unless \( P = NP \), no polynomial-time layout algorithm can ensure that the number of tracks it requires never exceeds \( k \) plus the optimum, for any constant \( k \).
4. MODEL PROPOSAL

From the conclusions drawn in the previous chapter it is clear that none of the described area estimating schemes fully provided the features we need for our system. For that reason we have to develop our own approach that must satisfy as many of the features, we derived earlier, as possible.

To cope with the great complexity of estimating the wire length distribution in real physical layouts and to provide for a possibility to use earlier estimates for the calculation of new costs the circuit will be represented by a model. By adopting a model in stead of considering the real physical configuration of the circuit the problem is simplified but nearly always at the expense of accuracy. Details often characterising the original configuration are left out in the model. However a good model will allow for a considerably smaller complexity in representing a large variety of circuits. Moreover, in many cases nothing or little is known about the circuit structure so that we are forced to represent reality by a model.

In the next section the model is further explained together with the motivation for its choice. As the hardware will consist of standard cells this design methodology on which our model is based, is briefly pointed out in the second section.

4.1 ASSUMPTIONS

The occupied area on a chip consists of the total module area and the wiring area. The wiring area is the area needed by the wires to interconnect the different modules. The total module area is simply the sum of the separate module areas. The dimension of a channel is determined by the length of the adjacent cell rows and the number of tracks needed for successfully routing the adjacent cell rows. The number of tracks needed largely depends on the placement of the modules. An optimal placement will yield minimal channel density and thus minimal chip area. For our area estimating scheme we do not want to do the actual placement as it will take too much processing time. Moreover, the estimates are needed to choose between different hardware configurations and above all things a good relative accuracy is required. Nevertheless we need some data on the wire distribution to estimate the space needed for interconnections. All this gives rise to the first two assumptions.

The hardware is modeled into a one-dimensional placement. The occupied area is the sum of the area required by the cells (i.e. the height of the cells times sum of the module widths) and the area required for wiring the cells (see figure 4.1).

As only the relative accuracy of the estimates is important no folding of the 1-D placement is needed for modeling the placement.

The reasons for modeling the placement this way is that 1-D placement has been known to produce very good results (e.g. [Kang83] [Ueda85] [Schuler72] [Supowit83] [Feller78]) and it is fast method to obtain a circuit structure and thus estimates for wiring area.

It was a design objective to use a bit-slice layout methodology. Consequently, synthesising hardware is now possible by considering only one bit-slice and thus avoiding a lot of network data that is similar for each slice. The final design can be generated out of this single bit-slice. This bit-sliced approach leads to a third important assumption.
Figure 4.1. Required area for one slice.

Only a one-dimensional placement for one bit-slice is needed. The total area is the area occupied by one bit-slice times the width of the datapath.

Now that we have chosen for a 1-D placement, the task is to find a linear placement method that satisfies our object function. Turning to figure 4.1 it is obvious that the object function should result in a minimal track requirement. For the number of tracks determines the width of the top channel and thus has a great impact on the total width of the layout. In the next chapters this model will be further worked out together with its Common Lisp implementation.

4.2 BIT SLICES AND STANDARD CELL METHODOLOGY

As mentioned in the previous section the hardware may finally be constructed out of bit slices. In this technique functional elements as registers, multiplexers and operators are composed of bit-wise units. These bit-wise units, also called organelles are "glued" together in one direction to form word-length units, which are then bussed together in the other direction. Data paths of any desired width may be created using these organelles. Figure 4.2 shows how such a bit sliced structure is organized. The data path in the picture runs horizontally while control signals approach from below.

The organelles will be implemented using standard cells. This design methodology uses a library of predesigned cells. These cells usually have the same height, but different widths depending on the complexity of the function they have to realize. When placed together, the cells will abut vertically so that their power, ground (and sometimes global clock) lines will be connected automatically. Other signals are available on pins situated on top and bottom of the cells. In many cases cells are designed so that any cell signal is available on equivalent pins on top and bottom of the cell thereby giving the designer the freedom of connecting on the top or on the bottom. These cells are called double entry cells. The cells are arranged in rows of near equal sizes. The spaces between adjacent rows are called channels. Channels are used to route the signal wires between the cells.

Routing between non-adjacent rows can be achieved by using feedthroughs which consist of cells whose input and output pins are electrically equivalent. Feedthroughs are inserted in a row to permit multirow signals to be routed across the row instead of going around it. The placement procedure for standard cell layouts consists of assigning cells to rows (i.e. partitioning the design) and of assigning locations to cells within a row (i.e. finding a 'good' permutation of the cells in the row). In [Richard84] the most common techniques are described. It should be pointed out here
that in a bit sliced layout the cells only need to be ordered in one row. The feedthroughs in the latter layout style will nearly always be due to control lines.
5. MODEL DESCRIPTION

In this chapter the proposed model is worked out further. The linear ordering heuristic is described in detail and adaptations for incremental use are proposed.

5.1 PROGRAM ARCHITECTURE

The hardware generator scans the state graph level by level. At each level states represent different hardware implementations. All circuit and accompanying information is stored in these states. The cost calculating tool can be called from each state and at that moment operates on information characteristic for that state. The tool is unaware of other implementations belonging to other states as it has no storage function whatsoever. When costs are calculated for a new hardware implementation these are passed back to the hardware synthesizer together with updated model data. This model data is only of importance to the cost calculating tool. In the next it will also referred to as the cost environment. The cost environment is stored in the state. This cost environment, which enables for an incremental cost calculation, will be passed back to the cost-tool when new cost calculations are needed. Figure 5.1 shows a schematic of the iterative cost estimating program.

Figure 5.1. Program architecture.

The hardware generator provides the cost-tool with circuit changes together with the matching environment. The internal data formatting is done to process input data for following tools and sets up data structures needed for internal use. On the affected items of the new circuit a critical path analysis is done. In parallel the new placement for the adjusted circuit is constructed. From the newly obtained placement and delay the costs for the current circuit are derived. Finally the costs and environment data is passed back to the hardware generator.
5.2 CIRCUIT ALTERATIONS

During the hardware generation implementations for new demand graph nodes are inserted in the already synthesised circuit. To realize those insertions, two basic operations are necessary, the first the combination of adding and/or extending (in the following referred to as extension), the other the removal of circuit parts. With both operations all circuit changes can be covered. For reasons of efficiency, a third operation, the substitution was introduced. For a number of reasons explained later we may want to exchange an already present operator with an operator capable of realizing the same or even a larger set of functions. If the set of old terminal connections also fits on the new operator a substitution of the old operator for the new one is possible. The order of execution is determined as:

1. removals
2. substitutions
3. extensions

In the next sections these three operations will be further explained together with their implementation.

5.2.1 CIRCUIT EXTENSION

The most frequently occurring change to a circuit is the addition of circuit parts. There are basically four possible extensions to a circuit which will occur in combination with each other.

- a new net is introduced
- a new module is introduced
- an existing net is extended
- a new net is connected to a free terminal of an existing module

A new net will always be connected to new or already present modules. An existing net may also be extended with both new or already present modules. Circuit extensions must be passed to the input in a list, called addition of all affected and new nets (see figure 5.2).

![Figure 5.2. List of circuit extensions.](image)

This list has the same format as the actual net list describing the whole circuit. Nets are mentioned in their new configuration. Affected nets are nets that have been extended or that experienced a change in one or more of their connected modules. Because substitutions and deletions are dealt with separately the nets they affect should not be mentioned in the “addition” list. Nets to which components are added and deleted must be mentioned in both lists.

5.2.2 CIRCUIT REMOVALS

As an effect of both extensions and substitutions old hardware may loose its function. Connections
and modules which used to be essential in the data path may have been substituted by more efficient and powerful subcircuits. The superfluous parts are removed from the net list by the remove operation. The following cases may occur, often in combination with each other:
- a module is removed
- all or part of a net is removed

When a module is removed all connected nets are affected in that they lose one terminal. If one or a few terminals of a net are to be removed and the associated modules will not be removed, both net and modules lose terminals. The deletion of a whole net means that all connected modules lose terminals. Requests for removals are passed to the input in a list called deletions. This list contains all the nets that are affected by a removal together with all the modules that remain connected to the net (see figure 5.3).

5.2.3 MODULE SUBSTITUTIONS

In many cases it is favourable to reuse hardware already present in the circuit. In general the most optimal and thus smallest and cheapest module instantiations are used to implement functions. As a consequence it will often occur that earlier synthesized operators, multiplexers or registers do not meet the specifications needed for the current implementation of a function. Instead of implementing a new and more powerful module next to the first instantiation, the latter could be exchanged for a more powerful instantiation that also suits the current implementation step.

The main constraint for substitution is that the new module connects to all the wires that were connected to the substituted module before the substitution operation took place. This constraint guarantees that the circuit environment of the regarded module remains unchanged and makes the substitution an efficient operation. To meet the constraint it will often be necessary to combine the substitution with both removal and extension. A substitution request is passed to the input in a list called substitutions which lists the names of all substituted elements.
5.3 PLACEMENT AND RETRACTION

In this section the linear placement algorithm is pointed out. In the following sections issues related to this algorithm are further worked out.

5.3.1 BUILDING A LINEAR SEQUENCE

I chose the heuristic described by Kang to do the linear placement. It is straightforward and provides better placements than the already mentioned heuristic of Asano. In this section the algorithm will be described in detail.

5.3.1.1 PROBLEM DEFINITION AND STRATEGY

The problem is, given a circuit consisting of components (sometimes called modules, cells, elements, etc.) interconnected with nets, put the components in a linear sequence so that the number of nets cut by a plane separating two adjacent components is minimized. The concept used is basically equivalent to the net-cut model first described in [Schwiekert72].

Given a set of components, the algorithm selects one component at a time to build a linear sequence. The selection is done so that a selected component minimizes the net-cuts. During the process, the components are divided into three groups, i.e. \( IN \) set, \( ACTIVE \) set and \( OUT \) set. These groups are defined as follows:

\[
\begin{align*}
IN &= \{ i \mid i \text{ is a selected component} \} \\
ACTIVE &= \{ a \mid a \text{ is a non-selected component connected to an active net} \} \\
OUT &= \{ c \mid c \text{ is a component not in } IN \text{ nor in } ACTIVE \}
\end{align*}
\]

active nets are defined as

\[
active-nets = \{ n \mid n \text{ is net from item in } IN \text{ to item in } ACTIVE \text{ or } OUT \}
\]

The \( IN \) set contains the components already selected. The \( ACTIVE \) set contains the possible candidates for selection. The rest belongs to the \( OUT \) set. Thus the main process is to select a component in \( ACTIVE \) and move it to \( IN \) followed by properly updating both other sets. The \( IN \) set is initially empty and the \( OUT \) set becomes empty after final selection.

The nets coming out of \( IN \) are called active. Net-cut is the number of active nets. The components connected to active nets but not in the \( IN \) set are active. The set of active components is the \( ACTIVE \) set. Selection of active components is done from \( ACTIVE \) during the process. After each selection and move, all sets are updated properly.

Since \( ACTIVE \) is empty at the beginning, a component is selected from \( OUT \). It is called the seed. The seed is the most lightly connected one in the set. As will be shown in a later section with examples, selecting a proper seed is important to meet the objective of minimizing net-cuts. It is the seed selection which makes this ordering scheme unique. If \( ACTIVE \) becomes empty during the process, a new seed is selected in the same manner.
5.3.1.2 DESCRIPTION OF THE ALGORITHM

Based on the strategy described above, a linear ordering algorithm is presented. The detailed explanations on each step follow the description of the outline of the algorithm.

**ALGORITHM:**

\[
\begin{align*}
\text{OUT} & := \{ \text{all components} \} \quad \text{step 1} \\
\text{while} \ \text{OUT} \ \text{not empty} & \\
\quad \text{seed} & := \text{lightest component from OUT} \quad \text{step 2} \\
\quad \text{move seed from OUT to IN} & \\
\quad \text{move all newly active components from OUT to ACTIVE} & \quad \text{step 3} \\
\text{while} \ \text{ACTIVE} \ \text{not empty} & \\
\quad \text{select choice from ACTIVE that minimizes #active-nets} \quad \text{step 4} \\
\quad \text{move choice from ACTIVE to IN} & \\
\quad \text{move all newly active components from OUT to ACTIVE} & \quad \text{end} \\
\text{end} \\
\end{align*}
\]

At step 1, nothing is active. All components are in OUT, and IN and ACTIVE are empty.

Step 2 is the seed selection. The weight of a component is defined as the number of other components connected to it. A seed is the lightest one in the set. This choice is guided by the aim of reducing the number of net-cuts.

When a component selected from OUT or ACTIVE is put into IN it tears all connected modules that are in OUT to ACTIVE. Those connected components that were already present in ACTIVE stay there. See figure 5.4.a and 5.4.b.

![Figure 5.4. Set updating.](image)

The nets connected to a component present in ACTIVE can be divided in three classes obtained by the situation before this component is actually selected.
1. Nets that become newly active (new net).
2. Nets that keep being active (continuing net).
3. Nets which terminate the state of being active (terminating net).

Before the component is selected, a new net is a connection between the selected component in \textit{ACTIVE} and \textit{OUT} or other components in \textit{ACTIVE}. A terminating net is connected to only the selected component in \textit{ACTIVE}, but may have several connections to components in \textit{IN}. A continuing net has one or more terminals to modules in \textit{IN} and to at least 2 modules in \textit{ACTIVE} (figure 5.5).

![Figure 5.5. Different nets.](image)

After the component is selected and moved from \textit{OUT} or \textit{ACTIVE} to \textit{IN}, a new net brings in new components to \textit{ACTIVE} from \textit{OUT}, except when this net only has connections within \textit{ACTIVE}. A continuing net does not affect any set, but may change itself to a terminating net. A terminating net now completely belongs to \textit{IN} and is dropped from further consideration. This update which also occurs in the second while loop is done in step 3.

In step 4 a component is selected from \textit{ACTIVE} if it is not empty. A net gain is defined for a component in this group as the \textit{number of new nets minus the number of terminating nets}. The following rules are proposed by Kang for the selection of components from \textit{ACTIVE}.

1. First select a component with minimum net gain.
2. For a tie, select one with larger number of terminating nets.
3. For a tie, select one with larger number of continuing nets.
4. If tie again, select lighter one.

5.3.2 INITIAL SEED SELECTION

Kang defines the \textit{weight} of an element as the number of other elements connected to it. He proposes to choose the most lightly connected element as initial seed. But considering the object function of minimizing the number of active nets all the time, the element with the least number of
connected nets would be a better choice. To determine the importance of the initial seed I ran a number of experiments with several random logic circuits, varying in complexity from 31 modules and 32 nets up to 166 modules and 171 nets. These experiments show that the initial seed indeed has a very large influence on final placement results. The differences in required tracks and net area for different seeds are on average 32 and 36 percent. Details of these results are presented in table 1 and table 2 of Appendix B. As is shown in table 1 often close to optimal seeds are among the modules with the smallest number of connected modules (old seed), but in this group also very poor seeds were found.

Motivated by these disappointing results I have tried to devise a better initial seed selection rule. As no correlation could be distinguished between the primary module properties (#connected modules, #connected nets) and the final placement results, I tried to find one considering the total circuit.

If the seeds resulting in fine placements are situated on a map of the circuit diagram it is striking that they are not homogeniously distributed over this map, rather they appear to be grouped in several clusters. Now let the circuit be represented as a connection graph, the nodes representing the circuit modules the edges their interconnections. A linear sequence of the circuit modules can be represented in this graph by drawing a new line visiting the consecutive elements starting with the seed and finishing with the tail element of the sequence. Let the circuit be partitioned in clusters so that the number of nets passing from one partition to another is minimized. Many good placements represented as such in the graph have the property that the line selectively scans the clusters rather than randomly visiting them. In appendix C this is illustrated. Observation of the relation of this line and the partitions led to the next statement.

Statement 1:
When a line of a path connecting the consecutive elements of a good linear placement enters a partition it will not leave this partition unless all elements of the partition have been visited.

In a visual display of a good placement it is also easy to distinguish clusters and stragglers, elements that are difficult to assign to any partition. In figure 5.6 five clusters are placed. Element P2 is a typical straggler.

Figure 5.6. Linear placement of well structured circuit.

The consequences of the previous statement give rise for the next assertion.

So the problem now is to choose a first partition that minimizes the total net length. Once a
Statement 2:
Considering only the circuit partitions and not the individual modules, a good placement can be obtained by selecting the most lightly connected partition first and successively selecting the partitions that yield minimal net cuts.

When a partition is selected we only need to consider the elements contained in it. A seed has to be chosen from this group after which the rest of the modules can be placed according to the placement algorithm. Then a next partition must be selected. For this a refinement must be provided to the selection rules used for modules. From already placed modules active wires emerge. These nets may have terminals in several unselected partitions. Similar to the module selection scheme, it is possible to divide nets into terminating, continuing and new nets. Selection of a partition may also be done by comparing quantities of these nets, however the aspect that partitions do not necessarily need to be of equal sizes must be taken into account. In figure 5.7 is shown that it is better to select partition A instead of partition B which should normally have been selected according Kang's rules if partition S is the seed partition.

![Figure 5.7. Selecting circuit partitions.](image)

This upgraded linear placement algorithm is not suited for incremental use. However the knowledge that natural circuit partitions are of great importance to a fine placement is useful to predict the performance of a placement algorithm.

Demand graph nodes can be mapped on several hardware implementations, each of which may be considered as a strongly connected piece of hardware. Hence, they can be considered as circuit partitions. Each time a demand graph node is implemented a new circuit partition is supplied to the circuit. Assume for a moment that no hardware is ever used twice. Then a complete analogue exists between the circuit extension with a partition and with a single module. The added partitions may also have a strong connection to partitions at the tail of the placement or to other partitions somewhere in the placement. To maintain placement consistency partitions should be removed starting at the tail until all affected ones are unplaced. Consequently, a replacement can be done.

In real hardware synthesis however, an objective is to reuse existing hardware as much as possible. As a result large partitions will not remain the same during the whole synthesizing process. Individual modules or small groups will remain unchanged.

The selection of an initial seed on the module level is not satisfactory. The connectivity information associated with each module gives no insight in the total circuit structure. An hierarchical approach to the linear placement restores this lack of information. Kernighan and Lin [Kernighan72] have proposed a powerful graph partitioning algorithm that takes \( O(n^2 \log n) \) time. The
partitions obtained are of constraint sizes. The algorithm has already successfully been used by Breuer [Breuer77] for his min-cut placement algorithm.

Based on the importance of partitions I have devised another seed selection rule.

First select the module with the smallest number secondary nets.

For a tie, select one with a smaller number of primary nets.

A primary net of a component is a net directly connected to the module where as a secondary net is a net that is connected to the connected modules of the regarded component and that is not a primary net. The idea behind the revised selection rules is that modules that lie in the "center" of the circuit usually will have more nets around.

In appendix D a comparison is made between placements obtained with the original seed selection rule and results by applying the new rules. On average the new rules give an improvement of 4% in total net length and 6% in the number of tracks. The new placements obtained are still 8 percent worse than those resulting from the best seed.

5.3.3 GLOBAL NETS

The advantage of the Kang approach over the C-algorithm lies in the use of the special selection group for components. The anticipated better performance is achieved for a large number of circuits. However, difficulty arises when heavily connected or global nets exist in the circuit. With global nets we identify nets that interconnect different parts of the circuit or stated in terms of the partitioning approach of the previous section, nets that interconnect different module clusters. On the other hand, nets which are internal to clusters are called local nets. Figure 5.8 shows an example that is somewhat exaggerated to show the difference between global and local nets.

Now we examine how global nets affect the placement algorithm. If a net exists that is connected to a lot of circuit clusters then the selection of a connected component makes all the other connected components belonging to different clusters active. This means that ACTIVE will contain components of many different clusters. This jumbling of natural clusters may influence placement in a negative sense. Obviously large nets will pull a lot of components to ACTIVE and so slow down the selection process. On the other hand, nets interconnecting several clusters may impose a natural order.

If there is a net connected to all components, by selecting a seed, all components become active. However, since a net is connected to all components, its contribution to net-cuts will not be affected by any ordering. This implies that it is no use to consider this net in the process.
The impact of global nets on the average size of ACTIVE was revealed through an experiment. A net was defined global if more than a predetermined number of connected components were still not active, i.e. in OUT. Results are presented in appendix E.

So far the problems concerning global nets may seem of minor importance. They merely affect the selection time. The global nets gain importance when contemplated in relation with our yet to describe munch strategy. An extension to a global net, something very likely to happen, will cause nearly the entire placement to be destroyed and replaced. This would be wasteful. In order to save the program performance it is extremely important to identify these nets. There are several possibilities to handle them:

- These nets could be made transparent to the cost calculating. An obvious manner would be not to insert these nets in the net list passed to the input. Another to chop them in smaller entities and thus to prevent their negative impact.
- A detection mechanism could be devised to trace global nets.
- A third way, and perhaps the most efficient, is to mark these nets already during the synthesis process as in this phase more information about the circuit structure is available.

I have introduced a global variable called *global-netsize* that determines the minimum weight of a global net. Only lighter nets will be considered in the munch operation. Setting this parameter at a high value will hardly yield any performance increase. On the other hand a too small value will result in only minor munch operations and terrible incremental placements. Some additional tuning is required to achieve maximal performance.

5.3.4 UPDATE PLACEMENT

Earlier the need for an incremental cost calculation tool was expressed. As the hardware generation process is essentially incremental, calculation effort may be reduced by using part or all of the results obtained for previous implementations.

When new hardware is created on top of the already existing partial implementation, this new hardware has to be inserted into the linear sequence representation of the old circuit. The most simple and straightforward way is to place the expanded circuit from scratch. This is efficient when circuits are small because only little overhead is needed. But as the hardware synthesis process progresses and circuits become more and more complex this approach becomes uneconomic. In these cases it would be profitable to use costs and placement calculated for the preceding circuit. The additional storage and computing time is now amply compensated by incremental processing. Finally a new revised placement of components for the new extended circuit must be obtained, hence the need for placement updating.

Usually the supplementary hardware is connected to the existing circuit. This implies that new and extended nets can be connected to it, modules and nets may be added anywhere. Moreover, nets and modules can be removed, modules can be substituted for others having different area or envelope properties etc.. In detail, the old placement may be considerably shattered and may need a major update. And even after an update the issue of consistency arises. With consistency is meant that cost results for two implementations originating from two different implementation paths should be the same if these implementations are equal. Referring to the hardware generation example of chapter 2 the costs calculated for both comparable states should be the same.

To get to a better appreciation of the notion of consistency with respect to the problem addressed, the impacts of network changes on already existing placements are examined in the next. Updating strategies, taking into account this idea, are proposed. To contrast with these strategies some
examples of rather unfavourable updating strategies are presented.

5.3.4.1 FALSE STARTS

One possible approach could be to take care of any deletions and substitutions and modify costs accordingly. Next the new hardware can be placed at the tail of the old placement. The costs for the new subcircuit and extended old nets are determined and added with previous updated costs. The total is returned.

Still another method could be to do substitutions and deletions first, then place the new hardware at the tail of the old placement, and calculate new costs from scratch by going over the updated placement.

Both approaches ignore the fact that the cost of an implementation heavily depends on the linear placement.

5.3.4.2 CONSISTENT LINEAR PLACEMENT

As substitutions do not change the circuit environment they can simply be executed by changing the old module name in the sequence to the new name.

Deletions have a more severe impact on the placement, they change the circuit structure. Any deletion, net or module, will make previously connected modules lighter. Remembering the module selection rules of our adopted linear placement algorithm, a lighter module is preferred over heavier modules if their first three selection properties are equal. And when a net is removed from a module times even get worse. This will translate in a smaller net gain and therefore the module is even more likely to be selected earlier in the sequence.

For circuit extensions we first consider the case that a new module is connected to an existing module. The old module will grow heavier and therefore would never be chosen earlier in the sequence, rather later unless preceding modules also became heavier. As a consequence the new module is likely to be selected a position further from the head of the sequence than the already placed module it is connected with. A second impact comes from existing modules that receive new connected nets. These modules grow heavier and consequently increase their net gain. For this reason they have a very large probability to be selected later by a succeeding placement operation.

On the basis of these observations an updating strategy is devised that takes into account the disrupting circuit mutations:
ALGORITHM:

\[
\text{\textbf{OUT}} := \{ \text{all new components} \} \\
\text{\textbf{group}} := \{ \text{all affected existing components} \} \\
\text{while group not empty} \\
\hspace{1em} \text{remove component from tail of placement} \\
\hspace{1em} \text{if component in group} \\
\hspace{2em} \text{remove component from group} \\
\hspace{1em} \text{fi} \\
\hspace{1em} \text{move component to OUT} \\
\text{end;} \\
\text{order components in OUT at tail placement}
\]

The proposed method exploits the fact that modules connected to new modules or nets grow heavier and therefore have no chance of an earlier selection in the sequence than was the case during the last placement. Furthermore, if the expectation that new modules are often connected to pieces of hardware added in the previous synthesis step is justified, the module removal operation will often be limited to the tail, and thus reducing replacement time considerably. The alarming effects of deletions can be reduced by the actions induced by circuit extensions. As deletions are expected often to go hand in hand with circuit extensions the idea may be justified that circuit changes due to deletions will be embedded in the extension regions. Consequently deletions will not affect the circuit as much as previously feared.

Three updating strategies that take into account the model consistency are described below.

5.3.5 STRATEGIES

The first strategy follows all aspects of the previous proposal. Delete the appropriate module in the linear sequence and store all the modules in the circuit that are directly affected by these actions. Then exchange modules for substitutes, if any. From the list that represents the extensions, new and affected modules can be distinguished. Now, repeatedly elements are removed from the tail until all affected modules have vanished. To prepare for a consecutive placement operation all modules connected with active nets coming out of the munching sequence are collected in \textit{ACTIVE}, the others are placed in \textit{OUT}.

Observation learned that the first strategy tends to be rigorous. In general most of the removed elements returned at exactly the same place after replacement. Close observation learns that only those modules connected to a new net or whose nets have grown remarkably heavier get new places. They are selected later, what can be expected reminding the module selection rules. Hence, a modification of the previous method. Now, instead of removing all the affected modules only those are removed that connect to a new net.

Still another method to increase the speed of the commutating algorithm is to consider the number of tracks while munching the placement. The retraction is done in two scans, both considering the interval starting with the first affected module and running to the tail of the placement. In the first scan the first module position is located for which the number of actual tracks plus the number of new nets passing is equal to the maximum number of tracks required up to that place (see figure 5.9).

If no such module is found the new hardware is placed at the tail, otherwise the placement is munched up to the located module. It is clear that this method is unsuited to calculate costs used for deciding between two comparable implementations. Though they may prove useful to produce
AFFECTED COMPONENTS

SPACE REQUIRED TO ROUTE NEW WIRES TO COMPONENT "B"

WIRING SPACE

FIRST AFFECTED COMPONENT

Component Sequence

LAST COMPONENT TO REMOVE

TAIL PLACEMENT

Figure 5.9. Model with track profile.

estimates for different implementations.

5.4 COST CALCULATION

In calculating the costs for a linear ordering obtained by the previously described commuting algorithm there are a few interesting approaches to consider.

A first and most straightforward method is to calculate the implementation costs for the whole ordering from scratch each time the cost function is called. For this method only the old placement must be stored and returned by the hardware generator. As the component ordering part of the cost function has the largest complexity this method will not dramatically increase the running time. However if circuits grow very large the actual cost calculation time may exceed the time required for incremental ordering.

If apart from old placement also old implementation costs are returned incremental cost calculation is possible. Here we can distinguish two different approaches.

If only the final cost of the old implementation is returned this cost must be first reduced with the cost of the part of the old placement that is munching. Together with the following placement operation this reduced cost can cumulatively be updated. Finally the new placement and according cost are returned to the hardware generator. This method has the advantage that it requires only small additional memory space and its calculation complexity is a constant.

Each time a component is added to the linear sequence during placement the costs are cumulatively updated. If these cumulative costs are stored with the according components in the placement the burden of decrementing costs while munching may be left out. Here computational complexity is improved at the expense of storage requirement.

Accommodations for both incremental methods are build in. The first one only requires the storage of the old final costs and according placement while the second method also needs the costs associated with all the other components. Selection of a method is possible by flagging with unrequired input variables.
5.5 PROGRAM COMPLEXITY

As the cost function will be called very frequently during the hardware synthesis its computational complexity is important. Let us first determine the average number of connected modules to a selected component. The selection properties of these modules will need updating. Let a circuit be characterized by the following parameters:

\[
\begin{align*}
    n & \quad \text{number of nets} \\
    m & \quad \text{number of modules} \\
    t & \quad \text{average number of net terminals, i.e. net weight}
\end{align*}
\]

Consequently a module will have \( n \cdot t/m \) nets connected to it. Thus a selected component from OUT or ACTIVE on average will pull \( (t-1) \cdot n/m \) components to ACTIVE.

Now a selected component from ACTIVE on the average will affect

\[
\frac{1}{2} \cdot \frac{n \cdot (t-1) \cdot t}{m}
\]

other components in ACTIVE or OUT. The affected elements in OUT move to ACTIVE, so they all will need selection property updating. At step 4 of the ordering algorithm a component in ACTIVE is selected on the basis of these properties. Therefore the values of these properties have to be determined before the selection takes place. Only the affected components have properties that may have changed, and need updating. The complexity of the selection property updating function is determined as

\[
O\left(\frac{n}{m}, t^2\right)
\]

Consequently the order of steps to perform to update the properties of all affected components will be:

\[
\frac{n}{m}, t^2, \frac{n}{m}, t^2
\]

For circuits that have far more nets than components or that have many nets with a large number of terminals the algorithm will be slow. However on average \( n/m \) and \( t \) are small constants and thus average run-time will not be affected by the circuit size.

Selection is done among all components in ACTIVE and therefore is dependent on the average size of ACTIVE. To determine this size we use an expression for the estimated number of tracks \( \mathbb{E}(d(x)) \). For the derivation of this expression the reader is referred to chapter 6 where a probabilistic model for area estimation is described. The variables \( W, p \) and \( q \) are also explained there. On average \( 1/2 \cdot (t-1) \) components per active net reside in ACTIVE, so

\[
|\text{ACTIVE}| = \frac{1}{2} \cdot (t-1) \cdot |\text{active nets}|
\]
Expanding this geometric sequence and assuming $W$ very large yields

\[ I_{\text{ACTIVE}} = \frac{1}{2} \cdot \frac{(t-1)}{W} \cdot \frac{w}{p} \cdot q \]

For a circuit the total number of nets is usually strongly correlated with the number of modules. The number of modules on its turn determines the length of the linear sequence and thus $W$. Hence we may write

\[ n = a \cdot m = a \cdot b \cdot W \]

in which $a$ and $b$ are constants.

and with $t$ being constant and $\tau = \frac{1}{p}$ we find

\[ I_{\text{ACTIVE}} = \frac{\tau^2}{\tau - 1} \]

In [Donah79] the average wire length for a linear placement of a circuit with large partitions is found to be $\tau = C^v$, in which $C$ is the number of components in a good circuit partition and $v$ is Rent's exponent. Thus for the average size of ACTIVE we can write

\[ I_{\text{ACTIVE}} = C^v \]

The other steps in the linear ordering algorithm have simple linear complexity. The running-time of the placement algorithm therefore is proportional to $m \cdot C^v$. Figure 5.10 shows how run-time is related to circuit size. The circuits considered are described in table 4 of appendix D. The complexity of the retraction function largely depends up on the kind of circuit and the order in which this circuit is synthesized. As previously pointed out cluster based synthesizing is favourable.

5.6 CRITICAL PATH ANALYSIS

The hardware generator needs information about the time spent by signals to propagate from one register to another as this limits the clock frequency of the circuit. Only those signal paths in the logic affected by circuit modifications are considered, using the intermediate path delay values
stored with every module.

At the start of a clock cycle, values stored in the registers are put out. The delay of signals is defined to be zero at the register outputs. The signal passes through logic, is processed together with other signals and finally reaches a register input. The input signals of a module with more than one input usually do not arrive at the same time. Small time delays may be due to different lengths of signal paths which will not be further regarded here. Considerably larger delays are introduced by other modules. At each module a signal passes it suffers a certain delay. The total delay introduced by passing a module depends on the specific delay of the device added with the time the signal has to wait for other signals to appear at the other inputs. The maximum delay an input can experience is stored with each module.

Finally, all signals will reach the register inputs. The total signal delays are stored at the register inputs. The largest value of these input delays regarding all the registers determines the maximal clock frequency. The path traversed by this signal is called the critical path.

The removal of a module will affect the previous connected modules on a lower logic level. First the modules at the next logic depth are re-evaluated. This is done by checking the maximum value of their input signals. If this value is changed the modules connected at a lower logic level are considered. This process goes on until no maximum value changes any more or the signals have reached the registers. A substituted module will only affect its connected parts at lower logic levels. The same procedure described above is followed. Network extensions resort in more updating. First the signal delays at the inputs must be determined before any maximum delay is stored. These input delays are obtained from the maximum delay values stored with connected elements on a higher logic level. Subsequently, the earlier described process updates the maximal values of the lower logic levels.

After all maximum time delay values have been updated all the delay values of the registers are scanned and the largest delay found is returned.
6. PROBABILISTIC MODEL FOR AREA ESTIMATION

In this chapter a different area estimating scheme is presented based on a probabilistic model [Kurdahi85]. I have implemented and done some incremental experiments with this method. As the calculated cost for an implementation with this method was found to be a good indication of the circuit size this scheme seems suited for cost calculation. The fact that the model expects a value for the average connection length between its components, makes this method extremely suited for cost calculation of very large circuits in which this length is practically constant. Therefore an outline of the model is presented and an equation is presented to calculate the average wire length in a circuit assuming a linear array of well placed logic.

6.1 MODEL DESCRIPTION

The hardware will be laid out in bit slices consisting of standard cells. Not taking into account the control lines crossing the slice, the area of a slice is determined by the sum of the individual sizes of the used modules. The wiring channel area depends on the number of tracks needed to route the slice. Here we present a simple probabilistic model to estimate the track requirement. Without loss of generality we make the following assumptions about this bit sliced layout.

---
- The cells are arranged in one row.
- Cells have pins to which wires can be connected
- All cells have constant pin pitch which is defined as the distance between two consecutive pins or pin slots. The cell width is an integer multiple of the pin pitch. As a result the cell widths will be expressed in pin pitches. For example, a cell may be five pitches wide and have two pins, one at pin position (slot) 2, the second at pin slot 4. If the pin pitch is taken as the unit length in which the cell library expresses the cell dimensions no conversion is needed.
- All nets are two point nets (i.e. connecting exactly two pins). In real physical circuits nets connect often more than two modules, hence the need for modelling multi terminal nets as two pin nets.
---

Figure 6.1 gives a view of the row model.

![2 point net](image)

Figure 6.1. The row model.

As a consequence of the previous assumptions the module sequence is transformed into a row of pin slots where the distance between two adjacent slots is equal to the pin pitch. A pin slot is occupied if a wire emerges from it. The total number of pin slots in the row, i.e. the sum of the widths
of all cells, in pin pitches, is W. Further take the total number of two pin wires to be N.

For the next exposition we make the following additional assumptions:

— Pin slots are labelled 1, 2, ..., W from left to right.

— Wires travel from left to right. A wire starting at a pin slot is said to be born at that slot. The birth of a wire at a pin slot i is a random event with probability \(P_B(i)\).

— A wire born at a pin slot \(i\) will terminate (or die) at a certain pin \(j\) to the right of \(i\). This event is assumed to be dependent only on the distance between \(i\) and \(j\), \(i-j\), (which is also the length of the wire) and not on the individual values of \(i\) and \(j\). In other words, the length of a wire does not depend on where it is born.

— The length of a wire is assumed to be a random variable with a probability \(P_L(l) = \Pr(L=l)\).

![Figure 6.2. Density at point X.](image)

We define the density at a point \(x\) to be \(d(x)\) as the number of wires crossing a vertical cut line at \(x\) as shown in figure 6.2. The objective is to estimate the average value of \(d(x)\). To do that we have to look at the following events:

\[A(i) = \text{A wire is born at } i, \text{ and }\]

\[B(i, x) = \text{A wire crosses } x \text{ given that it is born at } i.\]

The probability of the first event is given by \(P_B(i)\). Now a wire born at \(i\) will cross \(x\) iff its length \(l\) is such that

\[x - i \leq l \leq W - i\]

So,

\[\Pr(B(i, x)) = \Pr(x - i \leq l \leq w - i) = \sum_{l=x-i}^{w-i} P_L(l)\]

Now, the probability of a wire being born at \(i \leq x\) and crossing \(x\) is \(\Pr(A(i)) \cdot \Pr(B(i, x))\). So the average number of wires crossing \(x\) is given by
The number of tracks needed is equal to the maximum density throughout the row. This means that, in order to estimate the number of tracks, one must estimate the average value of the variable \( d' = \max_x d(x) \). This, however, may be a very difficult task, especially for arbitrary forms of \( p_l(i) \) and \( p_b(i) \). One way of approximating the average value of \( d' \) is to use \( \max_x E\{d(x)\} \). It was actually shown (Sastry 85) that

\[
\max_x E\{d(x)\} \leq E\{\max_x d(x)\}
\]

This means that the approximation is actually a lower bound on \( E\{d'\} \).

The next task is now to make realistic assumptions about \( p_b(i) \) and \( p_l(i) \). The most general assumption for \( p_b(i) \) is to assume that pins are uniformly distributed among \( W \) slots. Hence for \( N \) wires, the probability of a wire being born at \( i \) is

\[
p_b(i) = p_b = \frac{N}{W}
\]

The choice of \( p_l(i) \) has been discussed in the literature on wiring space estimation and many distributions were suggested, such as the Poisson, exponential and geometric distributions. For the wire length distribution of row oriented placements the geometric distribution was found to be most suited (Gama81). This also corresponds with results I obtained from placements of random logic (Table 5 Appendix E). Therefore we assume that \( p_l(i) \) is geometric.

So, \( p_l = p q^{-1} \), where \( \frac{1}{p} = \text{average wire length} \) and \( q = 1 - p \). Given these two forms of \( p_b(i) \) and \( p_l(i) \), it is possible to obtain a closed form for \( E\{d(x)\} \)

\[
E\{d(x)\} = \frac{N}{Wpq} (1-q^x)(1-q^{W-x+1})
\]

Now, in order to find the maximum of \( E\{d(x)\} \), we set \( \frac{d\{E\{d(x)\}\}}{dx} \) to zero and solve for the \( \text{max}(x) \), so

\[
\frac{N}{Wpq} \left(-q^x \log q(1-q^{W-x+1}) + q^{W-x+1} \log q(1-q^x)\right) = 0
\]

or,

\[-q^x + q^{W-x+1} = 0\]

So, the maximal local density occurs at \( x_{\text{max}} = \frac{W+1}{2} \), and the estimate for the track requirement is \( E\{d(x_{\text{max}})\} \).
6.2 AVERAGE WIRE LENGTH

Several authors were concerned with the problem of a priori length estimation. Feuer [Feuer82J derives formulas for wire distribution, earlier mentioned in chapter 3, and for the average connection length of wires in a region of random logic. Donath [Donath79J [Donath81J obtains the same results, with a different approach. He derives an equation for an upper bound on expected average interconnection length of a linear placement of gates based on partitioning results. Actual placements are reported to give an average interconnection length of about half the upper bound predicted by his theory. The average wire length in terms of average partition size \( C \) is found to be

\[
\bar{r} = \frac{5}{3} \cdot \frac{(1 - 4^{p-1}) \cdot (C^p - 1)}{(1 - C^{p-1}) \cdot (4^p - 1)}
\]

in which

- \( C \) is the number of gates
- \( p \) is Rent's exponent, a small positive number originally fixed at \( \frac{3}{4} \).

This expression may already be applied for relatively small circuits as no assumptions are made about circuit size. Donath published results of circuits varying from 60 to 2148 gates. Rent's exponent accounts for the average number of connections per module. By partitioning the logic the average number of external connections of a partition can be obtained, from which the exponent \( p \) may be calculated from the empirical relation formulated by Rent [Russo71J.

\[
p = \frac{\log 1 - \log b}{\log C}
\]

in which

- \( I \) is the average number of external connections in well-partitioned logic
- \( b \) is the average number of terminals per module

Since highly parallel circuitry has large exponents \( p \) (as much as 0.75), while highly serialized circuitry may be designed with low exponents (as little as 0.47), this indicates that the wiring space requirements may vary drastically as the character of the logic changes.

6.3 REMARKS

The presented model may be used if a value for the average wire length is available. This can be derived from linear placements or from partitioning the logic to get estimates for Rent's exponent. Kang's algorithm takes \( O(n \log n) \) time, The Ford and Fulkerson and Kernighan and Lin partitioning algorithms resp. take \( O(n^2) \) and \( (n^2 \log n) \) time. It must be pointed out that the value calculated for \( p \) will be unpredictable for small circuits. But as the amount of hardware increases this partition exponent will stabilize and will not need frequent updating any more.
7. PROGRAM IMPLEMENTATION

In this chapter some typical aspects of the model implementation are discussed together with some related issues as use of data types and experimental results.

7.1 ASSUMPTIONS

As the program is primarily concerned with constructing a linear placement and determining the total net length the following assumptions can be made without loss of generality.

- All terminals on a module are assumed to be situated in the middle of the module boundary.
- A net does not have more than one terminal per module.
- The total circuit may consist of several disjunct sub circuits.
- A net must be connected to at least two modules.

An extra and rather severe restriction on circuit structure is imposed by the critical path analysis function. The circuit must not contain any loops in which no register is inserted.

7.2 INTERFACES

To secure a smooth communication between the different program modules of the silicon compiler a uniform network description format is agreed upon. The network will be described by a Common Lisp data type called structure. This format will be used whenever network information must be exchanged. Currently a list of net and a list of module vectors (see figure 7.1) is used to transfer network information. These vectors must not be confused with those internally used by the cost function.

Apart from the final costs the cost environment returned by the cost calculation tool is only of interest for the latter. Therefore this data is stored through synthesis phase separated from the network description. The data structure of the cost calculation tool does not have this separation. The cumulative costs are stored in the module-vectors (see next section). This internal data format was determined long before the network description format was established. Hence, data interchange with the hardware generator is done through the use of interfaces.
7.3 INTERNAL DATA STRUCTURES

Circuits can be fully determined by their nets and associated connectivity information. All the information concerning the connections of a net is packed in a net-vector. So, with each net a vector is associated. The choice for a vector was inspired by the need for quick data access and reasons of flexibility. If more information is required the vector can simply be extended with the needed number of fields.

Internally a network is described in a net-duo-list, an association list consisting out of duos. The key of each duo is a net identifier, its value is the associated net-vector. For internal use a similar module-duo-list is maintained. Now the key of a duo is a module identifier and its value the associated module-vector. The syntax diagram of the net duo list is presented in figure 7.2.

Nets and modules are internally represented by symbols. The associated vector is the value of such a symbol. This is a very appropriate manner to store net and module properties. Referring to a vector field goes faster than to a property.

The completion of the internal net and module vectors is described in the chapter about implementation aspects. Appendix A provides a complete list of production rules.

7.4 INTERNAL VECTORS

The internal net-vector has four fields. In the next lists the defined global constant determining the index is mentioned first.

1. *netmodules* This is a list of all the modules connected to the net.
2. *netweight* This is the length of the previous list. It is made a separate entry as it is often needed.
3. *mods-with-input* List of all modules that are connected with an input terminal to this net.
4. *mods-with-output* List of all modules that are connected with an output terminal to this net. The lists are used to determine delay paths.

The internal module-vector has eight fields. They are:

1. *modconmod* This is a list of all the modules that are connected with a net incident to this module.
2. \textit{modnets*} 
   List of all incident nets.

3. \textit{where*} 
   Field for quickly determining the group in which module is placed.

4. \textit{cost-info*} 
   Five field vector for storing the cumulative costs.

5. \textit{in-nets*} 
   List of nets on input terminals.

6. \textit{born-signals*} 
   List of nets on output terminals.

7. \textit{maxdelay*} 
   Value that determines the maximum delay of input signals.

8. \textit{lib-name*} 
   The library name of the module.

Five cumulative cost variables recording different costs of the placement are stored with the components in the sequence. All cost variables are accommodated in a cost-vector. These five variables are:

1. A list of active nets together with the number of used terminals of the net.
2. A number for the cumulative module area requirement.
3. A number for the cumulative net area requirement. As nets are assumed to have their terminals in the middle of a module the required net area between two modules is determined as the number of active nets between both modules times half the sum of both module widths.
4. A number for the cumulative sum of module widths = sequence length
5. The maximum number of tracks needed to route the placement up to this module.

7.5 INPUT & OUTPUT

Internally all nets and modules are represented by a symbol with as value the internal vector. In the input and output list, modules and nets are represented by their name, i.e. a string. The input interface function changes this string representation into a symbol representation and interns these symbols. The output interface function first uninterns all symbols and changes their representation back to strings.

Figure 7.3 shows the syntax of both the sequence of input arguments and the output list. Further production rules can be found in appendix A. The list named changes contains the extension list, the deletion list and the substitution list in the same order. The argument costs appearing in input and output is a vector of five elements all of which are integers. The first is the maximum delay of the circuit. The second and third entry are respectively the total area occupied by the circuit components and the total area needed for wiring the linear placement. The last two numbers are respectively the total length of the latter placement and the amount of tracks required for routing.

The list called placement is the sequence of module names that is stored through the synthesis phase and returned at a new cost request. The next two lists have the same syntax as the second and third argument but describe the previous circuit implementation. They are required to determine the total impact of a circuit extension. The last argument called old-cost-vecs is a list of internal module cost vectors in the same order as the according modules appear in the placement sequence. The cost vector is the entry at index \textit{cost-info*} in the internal module vector and stores the cumulative costs.

In the output list the updated cost environment is passed back to the hardware generator.
7.6 FACILITIES FOR INCREMENTAL USE

The first two updating strategies described in section 5.3.5 are implemented. Only the first one is installed. The first one considers all the affected modules taking into account those connected to global nets. The second only considers modules that receive new nets on their terminals, i.e. grow heavier. Experiments still must be carried out to reveal the advantages and disadvantages of both methods.

Both incremental cost calculation mechanisms are implemented and installed. The first one only needs the final costs and placement sequence of the preceding implementation to calculate the new costs. It does not need the old list of cumulative cost vectors. The second method only needs the list of cost vectors. Therefore, if the input list placement is nil, and the list of cost vectors is not nil the second calculating scheme is used. On the other hand when the list of cost vectors is nil and the list placement is not nil, the first scheme is used.

It must be pointed out here that using the first calculation scheme implies that every critical path analysis must be done for the whole circuit as no intermediate signal delay values are available. However if no delay figures are needed this first scheme is favourable because of its smaller memory requirement.

7.7 EXPERIMENTAL RESULTS

For both structured and random logic circuits incremental cost calculations have been made. Only a few experiments with structured circuits were done, due to the lack of available circuits. But I mention the results because they agree with the earlier theoretically derived behaviour of the cost tool. The experiments were all done with the basic updating strategy. All affected components are removed from the old sequence, no global net accommodation is provided.

For a first experiment the circuits were randomly chopped in sub parts. These smaller entities served as the subsequent addition to the circuit under construction. It turned out that the total placement time for both kind of circuits was strongly correlated with the number of additions times the time to place the circuit in one part. This is exactly what we may expect for random additions. In these cases the existing linear ordering is often completely munched.
Another observation from the previous experiment is that the quality of the final placement for the structured logic, is less dependent on the order of supplying the additions than is the case for the random logic. It seems that structured logic is easier handled, i.e. clustered, by the applied ordering algorithm than the other type of logic. More experiments will be needed to find out, if this is a real feature of the model or just a coincidence.

The structured logic can easily be divided in strong interconnected sub parts or clusters. This often is difficult or even impossible for random circuits. Incremental ordering with these clusters as additions resulted in fast and good placements for the structured logic. Results for the random logic were not so downright. Some were fast and produced fine final placements while others showed just the opposite. Although I have not had the opportunity for further investigation, I have the impression that the order of supplying the additions is responsible for the differences in performance. If the clusters are provided so that new clusters only are strongly connected to the tail of the placement the placement will only be munched for a small part and replacement is needed for a small number of components. If clusters however are strongly connected to parts further from the tail more munching and replacement effort is needed.

Some incremental cost calculations were done with artificially chopped circuits. As can be expected it turned out that cluster based incremental growth of the circuit indeed yields the fastest results. With cluster growth is meant the subsequent extension of the partial circuit with strong interconnected circuit parts.

A disturbing characteristic of the proposed updating strategy may demonstrate itself if an existing sequence is largely munched. Under the influence of the changed circuit, the replacement operation following the retraction sometimes yields a better placement (required net area, #tracks) than the previous one. Due to this, the according cost for the extended and larger circuit may well be less than for the preceding smaller circuit. This undesired behaviour points out that the relative accuracy is not automatically guaranteed and that updating strategies must be carefully designed. I met the previous shortcoming by introducing the concept of global nets in the retraction strategy, where after the malfunction did not appear any more.

An other additional method could be, to use only one initial seed for all following placements. Often the initial seed in a placement is affected by circuit changes and consequently is removed as the last element of the sequence. Due to the changed circuit structure and other ordering in OUT another initial seed may be chosen where after a new linear ordering is build up. As table 1 of appendix B shows, a different initial seed may have a large impact on final placement cost.

On the other hand an unconstrained munching strategy is expected to return the best and most consistent placements although no experiments were done to state this idea.

With the described probabilistic model some incremental cost calculations were done in which the estimate on average wire length was derived from a linear placement obtained by Kang’s linear ordering heuristic. The calculated cost were clearly proportional to the growing circuit size but also to the estimate of average wire length.
8. SUGGESTIONS FOR FURTHER RESEARCH

To increase the speed of the current program the internal data structure must be reconsidered. The interaction between hardware generator and cost calculation function must be tuned so that superfluous data type conversions are prevented. Instead of using the so called duo list, dotted lists could be used in association lists to avoid unnecessary access operations.

The assumptions made for the model may turn out to be too restrictive. They may be relieved but this will require an adaptation of the data structure and of most of the program routines.

To measure the performance and to get insight in the reliability of the cost calculation tool, still a lot of tests must be run with different classes of circuits. No extensive runs have been made yet to compare the various updating strategies. The impact of the proposed strategies can be used to decide on which strategy to use in combination with what type of circuit. For an additional check the calculated costs could be compared with actual layout results to find out the types of circuits the tools performs best with.

The hierarchical approach may also be further investigated. It may be favourable to detect clusters early in the synthesis, for instance in the demand graph. The intermediate group in the placement algorithm could then be filled with entire clusters instead of single modules. This will probably result in an improved linear ordering algorithm.

The probabilistic model presented can be further developed and tested. If an acceptable mechanism can be devised to derive Rent's exponent for a circuit, area estimation may be done independent from heuristically obtained linear orderings of components.
Two different cost calculation schemes were described. The first heuristic scheme is an attempt to provide a quick and inexpensive solution to the problem of area estimation. This is of great importance to the dynamic programming approach used by the hardware generator. We have adopted an incremental approach and tried to make plausible that this method can yield acceptable results. These statements were validated with experiments.

The model is based on a linear ordering of the circuit components and allows for incremental cost calculation. The best performance is obtained from structured logic circuits. This was pointed out in the report and was experimentally demonstrated.

Incremental ordering of random logic is slow as existing orderings are often largely munched. Moreover, the consistency of an incrementally obtained ordering often grows worse as the number of iterations increases. On the other hand it was observed that incremental placement of structured logic yields good orderings and that iterations are fast when a hierarchical e.g. cluster based approach is applied. Supplying circuit clusters in a natural order determined by the final circuit yields the best results. On average the existing placement only needs minor updates at the tail.

Real consistency in the sense determined in this report is only guaranteed if placements are done from scratch each time results must be compared. Moreover modules must be selected in the same order to establish equal costs. But relying on results from experiments done so far, the conclusion is justified that incremental cost calculation with the new heuristic model proposed in this report is possible. Nevertheless additional tuning of the updating strategies remains necessary.

The second proposed cost calculation scheme is of a theoretical nature. If a good estimate for the average interconnection length of the circuit can be derived the method is an alternative to estimate implementation costs. The few experiments all showed good results. This method may be found a good alternative to derive costs for large circuits in which the average wire length is fairly constant.
REFERENCES

Items are arranged according to the year of publication.

[Ford56] Ford, Jr., L.R. and D.R. Fulkerson
Maximal flow through a network.
and
Flows in networks.

[Gomory61] Gomory, R.E. and T.C. Hu
Multi-terminal network flows.

[Bellman62] Bellman, R.E. and S.E. Dreyfus
Applied dynamic programming.

[Kernighan69] Kernighan, B.W. and S. Lin
An efficient heuristic procedure for partitioning
graphs.

[Russo71] Landman, B.S. and R.L. Russo
On a pin versus block relationship for partitions
of logic graphs.

[Breuer72] Design automation of digital systems. Vol. 1:

[Schuler72] Schuler, D.M. and E.G. Ulrich
Clustering and linear placement.
In: Proc. 9th Design Automation Workshop, Dallas, Tex.,

[Schweikert72] Schweikert, D.G. and B.W. Kernighan
A proper model for the partitioning of electrical
circuits.
Ibid., p. 57-62.

[Adolphson73] Adolphson, D. and T.C. Hu
Optimal Linear ordering.

[Yoshizawa75] Yoshizawa, H. and H. Kawanishi, K. Kani
A heuristic procedure for ordering MOS arrays.
Leve, G. de
Leergang besliskunde. Deel 7a: Dynamische programmering 1.
MC syllabus, 1.7a.

Breuer, M.A.
Min-cut placement.

Goto, S. and I. Cederbaum, B.S. Ting
Suboptimum solution of the back-board ordering with channel capacity constraint.

Heller, W.R. and W.F. Mikhail, W.E. Donath
Prediction of wiring space requirements for LSI.

Feller, A. and R. Noto
A speed-oriented, fully-automatic layout program for random logic VLSI devices.
AFIPS conference proceedings, Vol. 47. P. 303-311.

Donath, W.E.
Placement and average interconnection lengths of computer logic.

Ohtsuki, T. and H. Mori, E.S. Ku, T. Kashiwabara, T. Fujisawa
One-dimensional logic gate assignment and interval graphs.
Ibid., p. 675-684.

Donath, W.E.
Wire length distribution for placements of computer logic.

El Gamal, A.A.
Two-dimensional stochastic model for interconnections in master slice integrated circuits.
[Syed81] Syed, Z.A.
On routing for custom integrated circuits.
Copies available exclusively from Micrographic Dept.,
Doheny Library, USC, Los Angeles, CA 90089, U.S.A.
p. 3370-B.

[Asano82] Asano, T.
An optimum gate placement algorithm for MOS one-
dimensional arrays.

[Feuer82] Feuer, M.
Connectivity of random logic.

[Yehc82] Yeh, C.
The prediction of requirements of wiring space for VLSI.
University Microfilms Order No. DA8221796.

[Kang83] Kang Sungho
Linear ordering and application to placement.
In: Proc. 20th Design Automation Conf., Miami Beach,

[Ngai83] Ngai
Computational stochastic models for channel routing
track demand.
Pasadena, Cal.: Dept. of Computer Science, California

[Reingold84] Reingold, E.M. and K.J. Supowit
A hierarchy-driven amalgamation of standard and macro
cells.

[Richard84] Richard, B.D.
A standard cell initial placement strategy.
In: Proc. 21st Design Automation Conf., Albuquerque,


[Cheng85] Cheng Chung-Kuan
Decomposition algorithms for linear placement and
applications to VLSI design.
In: Proc. 18th Int. Symp. on Circuits and Systems,
Kyoto, Japan, 5-7 June 1985.

[Fuji85] Fujii, T. and T. Kikuno, N. Yoshida, H. Morikawa
A heuristic algorithm for one-dimensional gate
assignment.
Ibid., 1451-1454.
[Heller85] Heller, W.R.
Package planning in VLSI.

[Watanabe85] Watanabe, T. and H. Baba
A floor-plan design system for LSI layout.
Ibid., p. 9-12.

[Kurdahi85] Kurdahi, F.J. and A.C. Parker
Area estimation of VLSI integrated circuits.
Los Angeles: Dept. of Electrical Engineering Systems,
University of Southern California, 1985.
Technical Report CRI-85-05

[Sastry85] Sastry, S.
Wirability analysis of integrated circuits.
Copies available exclusively from Micrographic Dept.,
Doheny Library, USC, Los Angeles, CA 90089, U.S.A.

[Ueda85] Ueda, K. and H. Kitazawa, I. Harada
CHAMP: Chip floor plan for hierarchical VLSI layout
design.

[Veen85] Veen, A.H.
The misconstrued semicolon: Reconciling imperative
languages and dataflow machines.

Simulated annealing without rejected moves.

[LaPotin86] La Potin, D.P. and S.W. Director
MASON: A global floorplanning approach for VLSI design.
Ibid., p. 477-489.

Higher levels of a silicon compiler.
Department of Electrical Engineering, Eindhoven
University of Technology, 1986.
EUT Report 86-E-163

Synthesis of hardware structures. To be published.

[Deo87] Deo, N. and M.S. Krishnamoorthy, M.A. Langston
Exact and approximate solutions for the gate matrix
layout problem.
APPENDIX A

EXTERNAL DATA STRUCTURES

**Input-sequence**:
- changes
- new-net-vecs
- new-mod-vecs
- costs
- placement
- old-net-vecs
- old-mod-vecs
- old-cost-vecs

**Changes**:
- extension
- deletion
- substitution

**New-net-vecs**:
- sys-net-vector

**Old-net-vecs**:
- sys-net-vector

**New-mod-vecs**:
- sys-module-vector

**Old-mod-vecs**:
- sys-module-vector

**Costs**:
- delay
- cum-mod-area
- cum-net-area
- cum-seq-length
- tracks

**Placement**:
- module-name

**Old-cost-vecs**:
- module-cost-vector
substitution:

```
1 -- module-name -- 2
```

extension:

```
1 -- sys-net-vector -- 2
```

deletion:

```
1 -- delete-pair -- 2
```

sys-net-vector:

```
1 -- net-name -- net-in-modules -- net-out-modules -- 2
```

sys-module-vector:

```
1 -- module-name -- module-in-nets -- module-in-ctrl-nets -- module-out-nets -- module-lib-name -- 2
```

module-cost-vector:

```
1 -- cst-comp-name -- cst-comp-cost-info -- cst-comp-max-delay -- 2
```

output-list:

```
1 -- cost-list -- placement -- new-cost-vecs -- 2
```

new-cost-vecs:

```
1 -- module-cost-vector -- 2
```

cost-vector:

```
1 -- cross-list -- cum-mod-area -- cum-net-area -- 2
```

```
1 -- cum-seq-length -- tracks -- 2
```

cross-list:

```
1 -- net-name -- used-terminals -- 2
```

2 -- 3 --
INTERNAL DATA STRUCTURES

module-duo-list:

```
(1) module-duo ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) module-duo ----> 2
```

net-duo-list:

```
(1) net-duo ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) net-duo ----> 2
```

net-duo:

```
(1) net ----> net-vector ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) net ----> net-vector ----> 2
```

module-duo:

```
(1) module ----> module-vector ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) module ----> module-vector ----> 2
```

net-vector:

```
(1) netmodules ----> netweight ----> mods-with-input ----> mods-with-output ----> net-name ----> 1
   ^                        ^                        ^                        ^                        |
   |                        |                        |                        |                        |
   v                        v                        v                        v                        v
  (1) netmodules ----> netweight ----> mods-with-input ----> mods-with-output ----> net-name ----> 1
```

module-vector:

```
(1) modconmod ----> modnets ----> where ----> cost-info ----> in-nets ----> born-signals ----> maxdelay ----> lib-name ----> 1
   ^                        ^                        ^                        ^                        ^                        ^                        |
   |                        |                        |                        |                        |                        |                        |
   v                        v                        v                        v                        v                        v                        v
  (1) modconmod ----> modnets ----> where ----> cost-info ----> in-nets ----> born-signals ----> maxdelay ----> lib-name ----> 1
```

cross-list:

```
(1) net ----> used-terminals ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) net ----> used-terminals ----> 2
```

netmodules:

```
(1) module ----> module ----> 2
   ^                        ^
   |                        |
   v                        v
  (1) module ----> module ----> 2
```

netweight:

```
(1) integer
```

mods-with-input:

```
(1) module ----> 2
   ^
   |
   v
(1) module ----> 2
```

mods-with-output:

```
(1) module ----> 2
   ^
   |
   v
(1) module ----> 2
```
where:

in
act
out
buf

net:

symbol

used-terminals:

integer

net:

symbol

used-terminals:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:

integer

max-delay:
APPENDIX B

This section comprises the results from exhaustive experiments on seed selection. In a first experiment all circuit components were subsequently used as initial seed for Kang’s linear ordering algorithm. The resulting placements are judged on their net area and track requirement. The first column lists the eleven random logic circuits on which the experiments were performed. The second column displays the best results with respect to net area and tracks that were obtained and the third column lists the worst results. In column four the difference in percent of the previous two columns is calculated.

Seed selection is done according one or more module properties. Two of those properties are the number of primary connected modules and the number of secondary nets of the candidate. A primary connected module is a module connected to a wire that is incident to the candidate. A secondary net is a net not incident with the candidate but incident to a primary module of the candidate.

It is often the case that a few or many components have the same selection property value. Only one of them is chosen. In the last two columns the average seed is presented. Its proportional difference with the best seed is given, again with respect to the final placement result (net area).

TABLE I. Distribution of both seed selection groups with respect to final net area requirement.

<table>
<thead>
<tr>
<th>CIRC</th>
<th>OPTIMAL SEED</th>
<th>WORST SEED</th>
<th>DIFF</th>
<th>#PRIM MOD</th>
<th>#SEC NETS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AREA TR.</td>
<td>AREA TR.</td>
<td>%</td>
<td>%</td>
<td>%</td>
</tr>
<tr>
<td>alu1</td>
<td>177 7</td>
<td>193 9</td>
<td>4.5</td>
<td>.75</td>
<td></td>
</tr>
<tr>
<td>alu2</td>
<td>1807 21</td>
<td>21290 26</td>
<td>14</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>alu3</td>
<td>12117</td>
<td>1593 24</td>
<td>14</td>
<td>6.3</td>
<td></td>
</tr>
<tr>
<td>col14</td>
<td>462 10</td>
<td>744 18</td>
<td>15</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>dcl</td>
<td>285 10</td>
<td>324 12</td>
<td>7.6</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>red53</td>
<td>279 10</td>
<td>436 13</td>
<td>26</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>in7</td>
<td>1522 16</td>
<td>2230 19</td>
<td>16</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>risc</td>
<td>1193 12</td>
<td>1607 18</td>
<td>15</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>dcl17</td>
<td>1086 15</td>
<td>1438 24</td>
<td>6.4</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>dc2</td>
<td>1294 20</td>
<td>1506 26</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>spla</td>
<td>3004 26</td>
<td>3662 37</td>
<td>12</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 lists the results of actual placements done with both the old seed selection rule proposed by Kang and the new selection rule. Numerical values and proportional differences with the optimal seed are given.
TABLE 2. Performance results for two different seed selection rules.

<table>
<thead>
<tr>
<th>CIRC</th>
<th>#MOD</th>
<th>OPTIMAL SEED</th>
<th>NEW SEED</th>
<th>OLD SEED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AREA</td>
<td>TR.</td>
<td>NET AREA</td>
<td>TRACKS</td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>%</td>
<td>num</td>
<td>%</td>
</tr>
<tr>
<td>alu1</td>
<td>41</td>
<td>177</td>
<td>7</td>
<td>172</td>
</tr>
<tr>
<td>alu2</td>
<td>133</td>
<td>1807</td>
<td>21</td>
<td>2029</td>
</tr>
<tr>
<td>alu3</td>
<td>120</td>
<td>1211</td>
<td>17</td>
<td>1321</td>
</tr>
<tr>
<td>co14</td>
<td>75</td>
<td>462</td>
<td>10</td>
<td>469</td>
</tr>
<tr>
<td>dc1</td>
<td>44</td>
<td>245</td>
<td>10</td>
<td>291</td>
</tr>
<tr>
<td>ret53</td>
<td>49</td>
<td>279</td>
<td>10</td>
<td>356</td>
</tr>
<tr>
<td>in7</td>
<td>141</td>
<td>1522</td>
<td>16</td>
<td>1533</td>
</tr>
<tr>
<td>risc</td>
<td>157</td>
<td>1193</td>
<td>12</td>
<td>1470</td>
</tr>
<tr>
<td>dtk17</td>
<td>110</td>
<td>1086</td>
<td>15</td>
<td>1118</td>
</tr>
<tr>
<td>dc2</td>
<td>103</td>
<td>1294</td>
<td>20</td>
<td>1362</td>
</tr>
<tr>
<td>spia</td>
<td>166</td>
<td>3004</td>
<td>26</td>
<td>3267</td>
</tr>
</tbody>
</table>

num: numeric value, %: percentage
APPENDIX D

Table 3 shows results of experiments in which the global net size is a parameter. Components connected to a net connecting more components than this global net size are not moved to ACTIVE if one of the components is selected for IN. The effect on the average size of ACTIVE act is given together with the run-time and the final net area, a quality indicator for the resulting placement. In Table 4 some characteristics of the considered circuits are shown.

TABLE 3. Impact of net size on the run time and average size of ACTIVE.

<table>
<thead>
<tr>
<th>CIRC</th>
<th>NET WEIGHTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>act</td>
</tr>
<tr>
<td>*alu1</td>
<td>4.4</td>
</tr>
<tr>
<td>*alu2</td>
<td>7.1</td>
</tr>
<tr>
<td>*alu3</td>
<td>6.3</td>
</tr>
<tr>
<td>*col14</td>
<td>7.3</td>
</tr>
<tr>
<td>*col1</td>
<td>5.7</td>
</tr>
<tr>
<td>*col5</td>
<td>4.6</td>
</tr>
<tr>
<td>*col2</td>
<td>7.3</td>
</tr>
<tr>
<td>*dk17</td>
<td>4.5</td>
</tr>
<tr>
<td>*risc</td>
<td>3.9</td>
</tr>
<tr>
<td>*in7</td>
<td>10</td>
</tr>
<tr>
<td>*apia</td>
<td>12</td>
</tr>
</tbody>
</table>

TABLE 4. Distribution of netweights for eleven benchmark circuits.

<table>
<thead>
<tr>
<th>CIRC</th>
<th>#MOD</th>
<th>#NETS</th>
<th>SUM</th>
<th>#NETS WITH NETWEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>*alu1</td>
<td>22</td>
<td>33</td>
<td>61</td>
<td>17</td>
</tr>
<tr>
<td>*alu2</td>
<td>121</td>
<td>125</td>
<td>246</td>
<td>92</td>
</tr>
<tr>
<td>*alu5</td>
<td>90</td>
<td>94</td>
<td>184</td>
<td>65</td>
</tr>
<tr>
<td>*col14</td>
<td>65</td>
<td>74</td>
<td>139</td>
<td>49</td>
</tr>
<tr>
<td>*col1</td>
<td>37</td>
<td>37</td>
<td>74</td>
<td>23</td>
</tr>
<tr>
<td>*col5</td>
<td>44</td>
<td>46</td>
<td>90</td>
<td>33</td>
</tr>
<tr>
<td>*in7</td>
<td>117</td>
<td>131</td>
<td>248</td>
<td>84</td>
</tr>
<tr>
<td>*risc</td>
<td>131</td>
<td>126</td>
<td>257</td>
<td>92</td>
</tr>
<tr>
<td>*dk17</td>
<td>96</td>
<td>99</td>
<td>195</td>
<td>66</td>
</tr>
<tr>
<td>*col2</td>
<td>93</td>
<td>96</td>
<td>189</td>
<td>65</td>
</tr>
<tr>
<td>*apia</td>
<td>152</td>
<td>154</td>
<td>306</td>
<td>93</td>
</tr>
</tbody>
</table>
APPENDIX E

In table 5 experimentally obtained wire length distributions are compared with the geometric distribution proposed for the probabilistic model. As this geometric distribution is meant for 2-pin nets, the nets in the four circuits are also modeled as 2-pin nets. Therefore a n-pin net is split into n-1 nets. The results show that the experimental distribution has a stronger exponential character than the geometric distribution. This is due to a too large average wire length which spreads out the geometric distribution. This large value for the average wire length is strongly related to the nature of the 4 considered circuits which consist of random logic.

**TABLE 5.** Experimental and theoretical wire length distribution.

<table>
<thead>
<tr>
<th>WIRE LENGTH</th>
<th>#NETS WITH WIRE LENGTH IN FOUR CIRCUITS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>atu2</td>
</tr>
<tr>
<td></td>
<td>exp.</td>
</tr>
<tr>
<td>1</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8-10</td>
<td>13</td>
</tr>
<tr>
<td>11-20</td>
<td>27</td>
</tr>
<tr>
<td>21-</td>
<td>29</td>
</tr>
<tr>
<td>total</td>
<td>240</td>
</tr>
</tbody>
</table>
(171) Monnee, P. and M.H.A.J. Herben
MULTIPLE-BEAM GROUNDSTATION REFLECTOR ANTENNA SYSTEM: A preliminary study.

(172) Bastiaans, M.J. and A.H.M. Akkermans
ERROR REDUCTION IN TWO-DIMENSIONAL PULSE-AREA MODULATION, WITH APPLICATION TO COMPUTER-GENERATED TRANSPARENCIES.

(173) Zhu Yu-Cai
ON A BOUND OF THE MODELLING ERRORS OF BLACK-BOX TRANSFER FUNCTION ESTIMATES.

TECHNOLOGY MAPPING FROM BOOLEAN EXPRESSIONS TO STANDARD CELLS.

(175) Janssen, P.H.M.
FURTHER RESULTS ON THE MCMILLAN DEGREE AND THE KRONENCKER INDICES OF ARMA MODELS.

(176) Janssen, P.H.M. and P. Stoica, T. Söderström, P. Eykhoff
MODEL STRUCTURE SELECTION FOR MULTIVARIABLE SYSTEMS BY CROSS-VALIDATION METHODS.

(177) Stefanov, B. and A. Veeffkind, L. Zarkova
ARCS IN CESIUM SEEDED NOBLE GASES RESULTING FROM A MAGNETICALLY INDUCED ELECTRIC FIELD.

(178) Janssen, P.H.M. and P. Stoica
ON THE EXPECTATION OF THE PRODUCT OF FOUR MATRIX-VALUED GAUSSIAN RANDOM VARIABLES.

(179) Lieshout, M.J. van and L.P.P.P. van Ginneken
OM: A gate matrix layout generator.

(180) van Ginneken, L.P.P.P.
GRIDLESS ROUTING FOR GENERALIZED CELL ASSEMBLIES: Report and user manual.

(181) Bollen, M.H.J. and P.T.M. Vaessen
FREQUENCY SPECTRA FOR ADMITTANCE AND VOLTAGE TRANSFERS MEASURED ON A THREE-PHASE POWER TRANSFORMER.

(182) Zhu Yu-Cai
BLACK-BOX IDENTIFICATION OF MIMO TRANSFER FUNCTIONS: Asymptotic properties of prediction error models.

(183) Zhu Yu-Cai
ON A BOUND OF THE MODELLING ERRORS OF BLACK-BOX MIMO TRANSFER FUNCTION ESTIMATES.

(184) Kadete, H.
ENHANCEMENT OF HEAT TRANSFER BY CORONA WIND.

(185) Hermans, P.A.M. and A.M.J. Kooi, I.V. Bruza, J. Dijkstra
THE IMPACT OF TELECOMMUNICATION ON RURAL AREAS IN DEVELOPING COUNTRIES.

(186) Fu Yinhong
THE INFLUENCE OF CONTACT SURFACE MICROSTRUCTURE ON VACUUM ARC STABILITY AND ARC VOLTAGE.

(187) Kaiser, F. and L. Stok, R. van den Born
DESIGN AND IMPLEMENTATION OF A MODULE LIBRARY TO SUPPORT THE STRUCTURAL SYNTHESIS.
(188) Jóźwiak, J.  
THE FULL DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE STATE AND OUTPUT BEHAVIOUR REALIZATION.  

(189) Pineda de Gyvez, J.  
ALWAYS: A system for wafer yield analysis.  

(190) Siuzdak, J.  
OPTICAL COUPLERS FOR COHERENT OPTICAL PHASE DIVERSITY SYSTEMS.  

(191) Bastiaans, M.J.  
LOCAL-FREQUENCY DESCRIPTION OF OPTICAL SIGNALS AND SYSTEMS.  

(192) Worm, S.C.J.  
A MULTI-FREQUENCY ANTENNA SYSTEM FOR PROPAGATION EXPERIMENTS WITH THE OLYMPUS SATELLITE.  

(193) Korsten, W.F.J. and G.A.P. Jacobs  
ANALOG AND DIGITAL SIMULATION OF LINE-ENERGIZING OVERVOLTAGES AND COMPARISON WITH MEASUREMENTS IN A 400 kV NETWORK.  

(194) Hosselet, L.M.L.F.  
MARTINUS VAN MARUM: A Dutch scientist in a revolutionary time.  

(195) Bondarev, V.N.  
ON SYSTEM IDENTIFICATION USING PULSE-FREQUENCY MODULATED SIGNALS.  

(196) Liu Wen-Jiang, Zhu Yu-Cai and Cai Da-Wei  
MODEL BUILDING FOR AN INGOT HEATING PROCESS: Physical modelling approach and identification approach.  