INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

***For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.**

UMI University Microfilms International

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8612731

Eichenberger, Peter Armin

FAST SYMBOLIC LAYOUT TRANSLATION FOR CUSTOM VLSI INTEGRATED CIRCUITS

Stanford University PH.D. 1986

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1986

by

Eichenberger, Peter Armin

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

University
Microfilms
International

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UM

Fast Symbolic Layout Translation for Custom VLSI Integrated Circuits

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Peter A. Eichenberger

March 1986

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Copyright 1986

by

Peter A. Eichenberger

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]
Dr. Mark A. Horowitz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]
Dr. Robert G. Mathews

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Oswald G. Villard
Dr. Oswald G. Villard

Approved for the University Committee of Graduate Studies:

[illegible]
Dean of Graduate Studies

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Symbolic layout tools have enormous potential for easing the task of custom integrated circuit layout by allowing the designer to work at a higher level of abstraction, hiding some of the complexity of full custom design. Unfortunately, the practicality of symbolic layout tools has been limited for several reasons. Most important, the CPU resources required to compute a full size integrated circuit from a symbolic description are prohibitively large; this problem has been avoided either by restricting the range of applicability to a narrow class of integrated circuits, or by using a simpler translation algorithm, which reduces the quality of the output. Other problems include: producing poor quality layouts, insufficient user control of the generated output, and inability to cooperate with other layout tools. There problems make symbolic design of complete chips difficult.

This thesis presents an approach to the symbolic layout problems that produces high-quality layout for an arbitrary circuit without requiring excessive CPU time. The key to this approach includes the use of hierarchy to improve CPU time, the use of wire-length minimization to improve quality, a good balance between optimization of the layout and optimization of CPU time, and a smooth transition over varying degrees of automation. The result has been a symbolic layout tool that has been successfully used to lay out several chips from a design-rule-independent input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I am grateful to the many people that provided help and inspiration for this work. In particular, I wish to thank Mark Horowitz, Rob Mathews and Oswald Villard for reading drafts of this dissertation and for providing valuable comments. I would also like to thank Rob Mathews and John Newkirk for the inspiration to work on these problems, and for their helpful suggestions and criticisms. Thanks go to Dan Perkins and Tim Saxe for their help in implementing Lava and much of the supporting software. Finally, I would like to thank my family and friends for their support and tolerance during this sometimes difficult time.

This work was supported in part by DARPA under research contract MDA 903-83-C-0335. The author's support by an IBM Fellowship during the 1979-1980 academic year is also gratefully acknowledged.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Illustrations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Introduction

Custom integrated circuits offer the highest performance, function density, and lowest manufacturing cost of various IC design techniques. These properties, in turn, allow smaller and less expensive devices to be built. The problem with custom ICs is that their design and verification is difficult, and, therefore, expensive.

One way to make IC design simpler, is to use a *semi-custom* design method such as *gate-array* or *standard cell*, where design consists of interconnecting a number of pre-designed subcircuits. This yields larger chips than full custom, where most of the pieces (subcircuits) have been designed to fit together for a particular chip.

Why is the process of designing a custom IC difficult? There are many reasons, although one of the major difficulties is the process of layout — that is, transforming an electrical description into geometry (the actual "picture" that will go on the masks at fabrication time). Layout is made difficult by a desire to minimize the area while obeying certain *design rules*. These design rules specify minimal distances between certain geometrical features to ensure manufacturability. For example, in Figure 1.1a, the distance between metal lines might have to be greater than 3μ (microns) to ensure they will not be shorted and the minimum required width of the metal lines might be 4μ to ensure that they are continuous. Another example of a design rule is in Figure

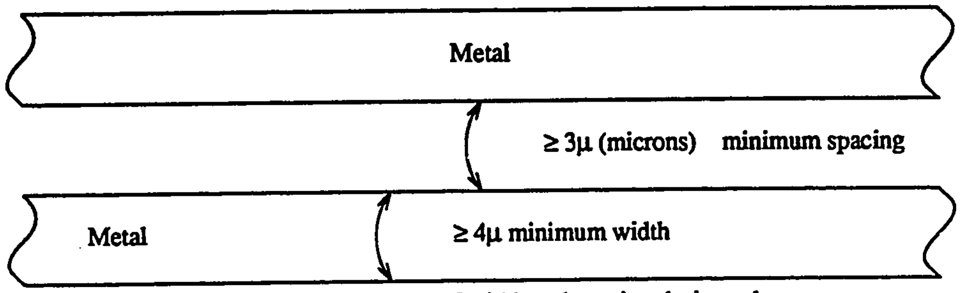


Figure 1.1a Examples of width and spacing design rules

1.1b, where the polysilicon line must extend beyond the gate of the transistor by at least 3μ to ensure the transistor does not become shorted. Many of the design rules also depend upon whether the objects involved are electrically connected or whether they are intended to be carrying different signals.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

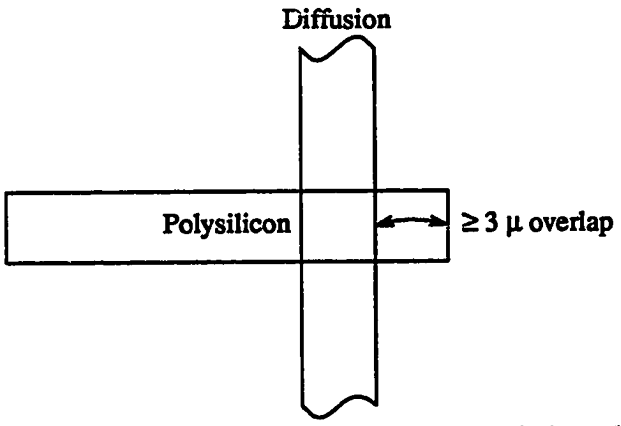


Figure 1.1b Example of overlap design rule

Simplified design-rules sets have about two dozen of these rules. Regular design-rule sets are 30 page documents. Constructing a layout that contains over a million objects and does not violate the design rules is a monumental task. Even worse is the task of synthesizing an IC to be of minimal size while obeying these rules. Design rule errors are inevitable.

How do we avoid working with design rules? The designer must be able to express his intent in a manner that has no implicit knowledge or assumptions about the design rules. This is precisely the intent of symbolic layout.

Symbolic Layout

In symbolic layout, the designer works with a notation that is representative of, but much more abstract than, the layout that finally appears on the chip. Typically, position information that would require knowledge of design rules is left unspecified, positions are approximate and relative, components and sometimes connection are referred to by names, etc.

The process of designing a custom IC consists of a series of transformations from a more to a less abstract specification[1]. Figure 1.2 illustrates a typical sequence of such transformations. Different design methods may break the problem into transformations in different ways. Each transformation is expensive and involves some "art" (for a high-quality result).

[1] At many of these levels, the specification is not a well defined machine readable notation, but is only an ambiguous human understandable notation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\begin{tikzcd}
\text{Architecture/Algorithms} \arrow[d] \\
\text{RTL (register transfer language)} \arrow[d] \\
\text{Logic} \arrow[d] \\
\text{Circuits: Electrical/Topological} \arrow[d] \\
\text{Geometric}
\end{tikzcd}$$

Figure 1.2 Design Transformations

The last transformation is probably the most tedious since it works with the largest data set. This transformation involves turning an electrical description with some topological information (approximate relative locations) into the geometry that will finally appear on the chip, taking into account the design rules and attempting to minimize the chip area. It should be the first to be mechanized, not only because it is probably the most expensive and tedious, but also is the easiest to mechanize well and will provide a basis for work on automating higher level transformations.

An example of a symbolic notation is *sticks* [Williams 78]. Sticks is a design-rule-independent layout description notation. In sticks, interconnect is drawn as narrow lines, positions are relative, transistors are represented as crossing polysilicon and diffisuion lines, etc. See Figure 1.3b for an example of a stick diagram for an inverter. Figure 1.3c is the layout represented by the stick diagram of 1.3b. Distances in the stick diagram are meaningless as the final dimensions will be computed from the design rules during translation. Sticks notation works best for describing low-level cells consisting of transistors, vias, and wires, but becomes awkward when extended to compose subcells into higher level cells.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

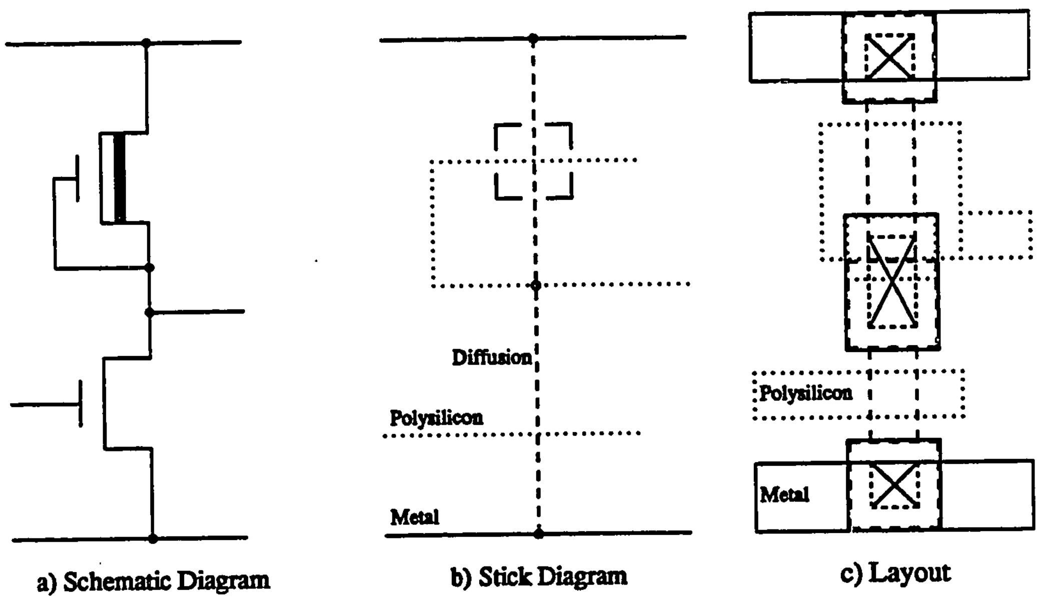


Figure 1.3 An NMOS Inverter

Symbolic Layout Translation

Some of the major difficulties of symbolic layout translation for entire ICs are caused by the immense size of the task. Also, when composing a large chip, the designer wishes to describe his layout to the translator in different terms than he would for a lower level cell. Efficiently implementing these higher-level translations is a distinct task.

Translation of stick diagrams or other forms of symbolic notation was attempted many years ago [Williams 78] [Hsueh 79], and much work continues in that area [Weste 81] [Buric83] [Dunlop 81] [Juran 83] [Kedem 84].

There are still problems, however, that have prevented symbolic layout from becoming a practical tool. Some systems are slow to translate even small cells and are impractical for translation of any significant portion of a chip. Others produce layouts of too poor quality to be used for designing serious chips. Most sticks translators are only designed to produce leaf cells (bottom level cells, which have no subcells), not entire chips, and therefore only produce maximally compacted cells. The user is left with the task of composing the compacted cells to form the whole

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

chip. Unfortunately, composing maximally compacted cells rarely produces a minimal sized chip, or even a small chip. In many cases, as in Figure 1.4 it is far more area efficient to compose cells by stretching and abutting them than to wire together mismatching subcells.

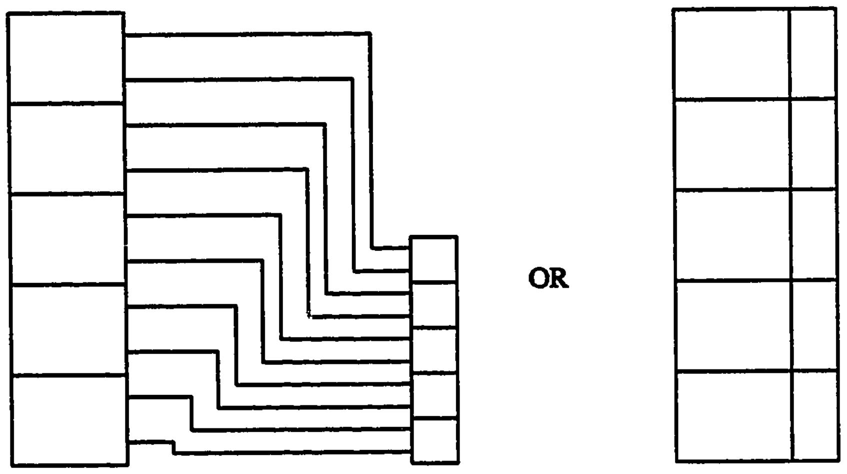


Figure 1.4 Advantages of stretch and abut.

CPU time is also an important consideration. Many algorithms are satisfactory for translating a single cell with a small number of elements in it. For an entire chip, which is a much larger problem, not only is the speed of the algorithm important, but also how the CPU time grows with the size of the chip (the asymptotic time complexity of the algorithm). CPU time that is linear in the chip size (area) would be ideal. Many translators' performance are far worse than linear because they contain algorithms that are quadratic, cubic, or even exponential in the worse case.

The quality of the output is another important consideration. While the notion of quality is hard to quantify, suffice it to say that merely producing any correct layout is not sufficient. The translator must produce a good layout in terms of area, wire length, *etc*., though not necessarily the best correct layout.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

My work

This dissertation presents a method to make symbolic layout of entire chips practical. While there is no single approach to achieve this goal, I have made a sequence of design decisions that appeared to lead most directly to a practical symbolic layout translator, and have made a number of measurements along the way. Many of the alternatives appeared to have significant problems, but now that one successful course has been charted and measured, exploring these other avenues more carefully should be much easier. In order to test many of the design decisions, and the algorithms they required, a system was implemented that could be used by designers of integrated circuits. This design system, called *Lava*, has been used by designers who had no part in implementing it.

The next chapter discusses previous related work in symbolic layout translation and solution of optimization problems. Chapter three begins by discussing the goals for an IC description method, then discusses the major features of the new IC description language Lava. The summary is a description of a partially compiled intermediate form upon which the algorithms of the next chapter are based. Chapter four describes the algorithms for generating systems of constraints and optimization problems from the intermediate form. Chapter five characterizes the optimization problems, then presents solution methods based on the characterization and analyzes their performance. Various tradeoffs and alternatives for performance improvement are discussed. Chapter six concludes the dissertation with a discussion of some of the major decisions made in designing Lava and the ramifications of those decisions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Background

This chapter discusses previous work relevant to this dissertation. The first section covers symbolic layout and layout compaction. Many approaches to the layout compaction problem reduce the task to a general mathematical optimization problem. These problems are not unique to the field of IC design automation, but have been studied by many other researchers. The second portion of this chapter deals with the solution of these optimization problems.

2.1. Layout Automation

One of the more tedious and error prone tasks in custom IC design is layout. For well over a decade, researchers have tried to automate this phase of IC design.

Symbolic layout began as simply a shorthand notation for representing layouts: easy-to-write symbols were used instead of combinations of carefully drafted rectangles [Gibson 76], [Barnes 75]. These symbols were drawn on a courser grid than the layout would have been, resulting in some waste of area. The course grid spacing was chosen to ensure the layout contained no design rule errors.

The loss of area due to the course grid can be mitigated by performing automatic *spacing* or *compaction* on the layout. Akers proposed IC-mask layout tools in a style similar to printed circuit board layout, but with some expansion and compaction features [Akers 70].

Sticks

Williams coined the term *sticks* to represent a kind of symbolic notation that was similar to a sketch the designer might draw of the circuit before laying it out [Williams 78]. The stick diagram is somewhere between a schematic diagram and a layout: relative placement of components and mask layers for interconnect have been chosen, but actual positions and dimensions have not been chosen. He also proposed a program to translate the stick diagram to rectangles and perform the compaction to produce a design-rule-correct layout. For instance, an NMOS inverter would be represented as in Figure 2.1b, while the schematic and layout are as in Figure 2.1a and Figure 2.1c.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

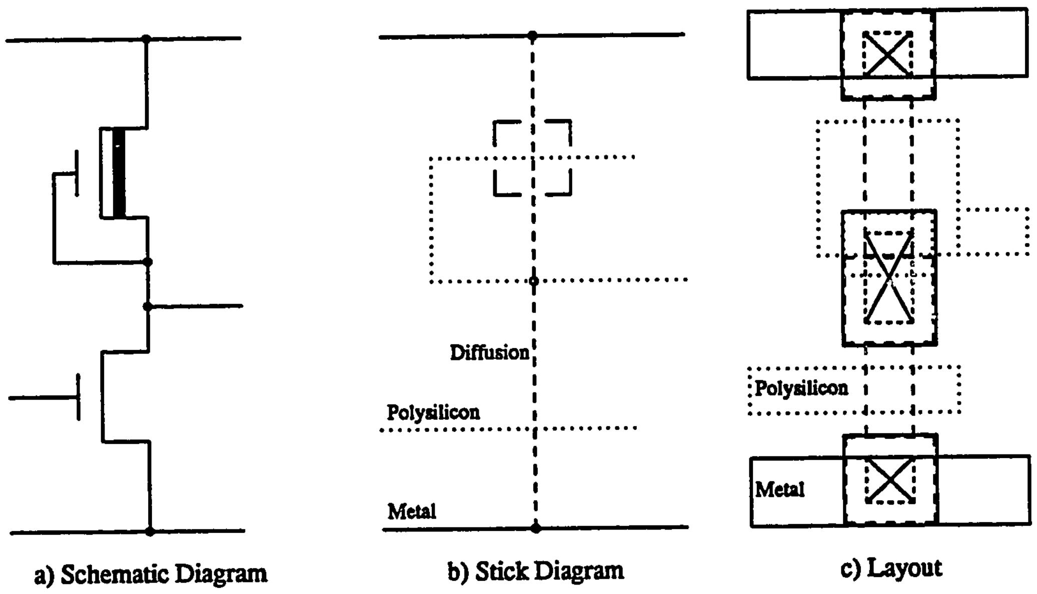


Figure 2.1 An NMOS Inverter

Layout Compaction

The task of layout compaction is to minimize or reduce the area of a layout (or portion thereof) while obeying the design rules and maintaining the integrity of the circuit. Although reducing the area to the absolute minimum is a very difficult problem for any significant layout, simply reducing the area to close to the optimum is practical.

The reason that optimal area reduction is so difficult is that the search space is not convex[1], meaning that it is not always possible to move smoothly and directly from one legal set of positions to another without passing through illegal positions as in Figure 2.2. The non-convex search space means that there may be many local minima, which make finding the global minimum much more difficult.

A popular technique for compaction is to reduce or minimize each dimension separately, alternating between dimensions until no further improvement is made[2]. Such an approach (called

[1] Precisely, a convex set is defined as a set such that if $\vec{X}_1$ and $\vec{X}_2$ are in the set, then $\alpha\vec{X}_1+(1-\alpha)\vec{X}_2$, $\alpha\in[0,1]$ is in the set. Here, the vectors represent the positions of all the objects in the layout.

[2] Each one dimensional problem (as formulated) usually has a convex feasible space.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

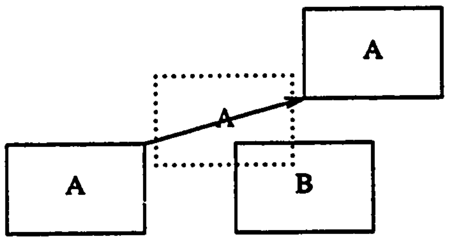


Figure 2.2 Non-convexity of compaction

one-dimensional compaction) frequently comes close to the best that may be achieved without reorganizations such as interchanging or rotating components. Another technique, called *two-dimensional compaction*, moves objects in both dimensions simultaneously [Kedem 84] [Wolf 84].

There are several approaches to layout compaction. One approach involves placing the mask features on a grid or other data structure representing the allocation of chip area, and directly manipulating these features. Another involves generating a system of inequalities representing the design rules and the relationships among the mask features. This system of inequalities, which can be represented as a directed graph, is solved to yield a set of compacted positions for the mask features. In many cases, the geometry manipulation algorithms can be described in terms of graph manipulations.

Shear Line and Virtual Grid Compactors

Dunlop, in SLIP, places the mask features on a course grid, and then searches for a sequence of grid points crossing the chip (the *compression ridge*) that are either adjacent, or connected by a *shear line*, that can all be eliminated without changing the circuit [Dunlop 78]. Shear-line compactors, in theory, are capable of performing almost as good a one-dimensional compaction as the coarseness of the grid allows. Unfortunately, the search for the removable grid points involves exploring many avenues and backtracking, and therefore can be time consuming. As the search time is a function of the number of grid points, a faster compactor requires a coarser grid and therefore can waste more area.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In virtual grid compactors, such as MULGA [Weste 81] [Boyer 83] or VIVID [Rosenberg 84], the layout is densely placed on a grid without regard to design rule spacing. The grid is then scanned to determine what spacing between each pair of grid lines is necessary to satisfy the design rules. For larger layouts, there are more unrelated components on a grid line, yet all must be moved together to account for the worst case spacing and cannot be moved relative to each other. For example, a virtual grid compactor could not achieve the following compression (in x), because "A" and "B" are on the same virtual grid line:

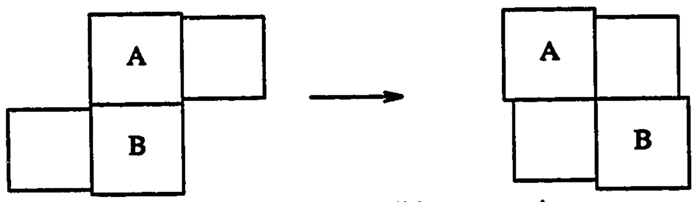


Figure 2.3 An impossible compaction

This inflexibility of virtual grid compactors can exact a large area penalty in large layouts. A more flexible compactor might be able to produce the drawing on the right.

Constraint System Compactors

One of the earlier compactors to build a system of inequalities was Hsueh's CABBAGE [Hsueh 79]. In Cabbage (as in many others), the constraints (inequalities) arise only from design rule spacing constraints as in Figure 2.4. Because the only user constraints are lower bounds on distance between features, Cabbage's constraint graph is acyclic and can therefore be solved for longest paths in time linear in the number of edges. Upper bound constraints are only checked, but not used to determine the solution.

A penalty of acyclic constraint graphs is that connections must be made to fixed locations on a component rather than being allowed to slide over a larger permissible area as in Figure 2.5. Hsueh uses a form of automatic jog introduction to mitigate the area costs of this rigidity.

Similarly, Mosteller in REST builds an acyclic constraint system (again, except for user-provided upper bound constraints), but does no automatic jog insertion [Mosteller 81]. Dunlop in SLIM [Dunlop 81] also builds an acyclic constraint graph, but does automatic jog insertion and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

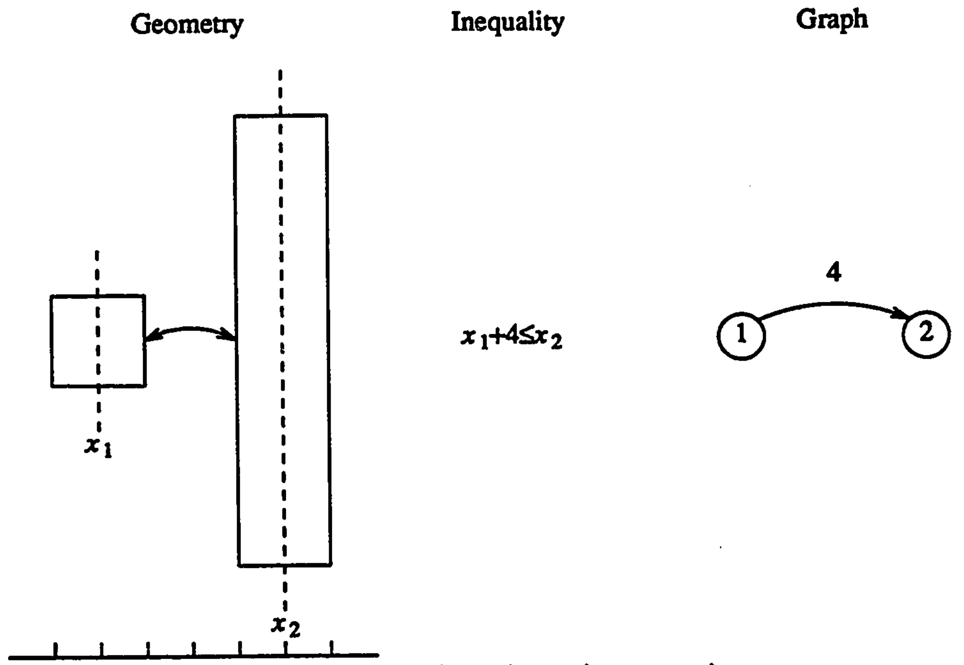


Figure 2.4 A design-rule spacing constraint

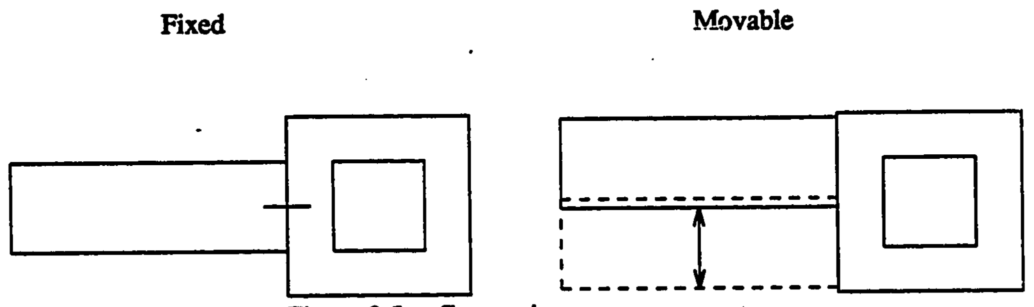


Figure 2.5 Connecting to a component

uses a variety of methods for reducing the difficulty of compacting large layouts. McGarity claims area performance of SLIM is poor [McGarity 83]. ALI uses a longest-path constraint graph where all edges have a non-negative length [Lipton 82]. Such a graph cannot have non-trivial cycles and is solvable in linear time.

Both Schiele and Kedem use cyclic constraint graphs to represent flexible connections [Schiele 83], [Kedem 83] [Kedem 84]. Schiele is also minimizing wire lengths in his compaction algorithm.

Kedem is trying to minimize area by compacting in both x and y directions simultaneously using an integer programming formulation to couple the problems. This simultaneous x and y

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

compaction is called two-dimensional or 2-D compaction, as opposed to 1-D, which compacts in each dimension sequentially. While 2-D compactions could potentially perform better than 1-D, Kedem offers no data comparing either performance (expected to be better) or CPU time (expected to be worse). Sastry shows that optimal 2-D compaction is NP-complete [Sastry 82].

2.2. Optimization algorithms

Most of these symbolic layout translators make use of the solution of some kind of mathematical optimization problem to perform the compaction operation. Most of these mathematical optimization problems have had applications in other fields than integrated circuit layout, and therefore have been well studied.

The graph-based symbolic layout tools compile the layout problem into a mathematical optimization problem, usually a longest-path problem. The longest-path problem is stated as follows: Given a set of inequalities

$$\left\{x_i+C\leq x_j\right\}$$

find the minimal difference between a pair of distinguished variables:

$$x_{sink}-x_{source}$$

The source and sink variables might correspond to the positions of the left and right edges of the cell being compacted, respectively (or bottom and top). This problem corresponds to finding the longest path in a graph from a source vertex to a sink vertex, where the arc lengths are the constants in the inequalities.

In practice, the result desired is not just the longest path from source to sink, or its length, but the length of the longest paths from the source to all other vertices. Fortunately, this information is what is most often computed by the longest-path algorithms, or it is easily computable in linear time from information available.

If the arc lengths are negated, this problem becomes the shortest path problem, an isomorphic, and more frequently studied problem. To be solvable, the graph must has no positive weight cycles (infinite longest path) and must be connected (no path).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acyclic longest path

The simplest of the optimization problems is the longest-path problem when the constraint graph is acyclic. This problem can be solved in time $O(v+e)$, where v is the number of vertices and e is the number of edges. These correspond to the number of variables and inequalities, respectively. The algorithm traverses the edges of the graph in topological sort order, which may also be determined in linear time, propagating the longest-path lengths onto the vertices.

Unrestricted Longest path

Slightly more difficult to solve is the longest-path problem where the constraint graph may have cycles. For there to exist non-trivial cycles, negative arc length must be permitted. Since the longest-path problem is a subset of the linear programming problem, the simplex method [Dantzig 63] could be used. As linear programming problems are quite a bit more general than longest-path, the simplex method is overkill and costs far more memory and CPU time than is necessary.

If an initial feasible (though not necessarily optimal) solution is available, it can be used to solve the problem in $O(e \log v)$ time [Johnson 73]. Johnson transforms all the variables (in the corresponding shortest-path problem) by the amount of the initial solution, resulting in a shortest-path problem with non-negative arc weights, and then uses a priority queue to traverse the edges in proper order to solve it. By making portions of his algorithm adaptive he is able to achieve a slightly better bound than mentioned above.

The Floyd-Warshall algorithm [Floyd 62] computes longest paths without an initial feasible solution, computes longest paths for all pairs of edges, and, unfortunately, requires $O(v^3)$ time.

Mathews used a relaxation-based algorithm to solve the longest-path problem [Mathews 81]. He found the time bound to be $O(B \cdot e)$, where B is the number of "back-edge" trees in the graph. Very similar are D. B. Johnson's "Arc set partition algorithms" [Johnson 73]. Liao uses a relaxation method that relies on the existence of an initial feasible solution [Liao 83].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A network flow formulation can be used to solve the longest-path problem. Kennington and Helgason describe a "Primal Simplex method on a graph" suitable for solving the network flow problem [Kennington 80]. Discussion of empirical performance of network flow as applied to IC layout may be found later in this dissertation.

Multiple longest paths

Some symbolic layout systems require the longest paths from multiple source vertices. If all vertices were used as source vertices, this problem would become the *all-pairs, longest-path problem.* The simplest, the Floyd-Warshall algorithm ($O(v^3)$) mentioned above, computes all pairs. D. B. Johnson has an all-pairs algorithm with a somewhat better time bound. In either algorithm, far more data than is needed by Lava is being computed, with the excess discarded. Mathews found it faster to compute individually the longest path from each source vertex, using results from the previous vertex to accelerate the speed of convergence [Mathews 81].

Reduction of wire length can be useful as a secondary objective during compaction. W. L. Schiele presents a method of reducing wire length [Schiele 83], although it is not clear whether a minimum is achieved. A method of minimizing wire lengths based on a network flow formulation is discussed later in this dissertation (§5.2).

Summary

In this chapter, I described some of the approaches that have been taken to automate the task of IC layout. First, a symbolic representation is used to ease the task of describing the layout. Because a direct translation of the symbolic representation leads to a waste of area, layout improvement called *compaction* is employed. Many of the compaction algorithms construct a system of linear inequalities and minimize some objective function subject to those constraints. In the second part of this chapter, I describe some of the existing optimization algorithms that are well suited to the optimization problems that result from symbolic layout translation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Lava: A New Layout Description Method

A new method of describing IC designs was developed to give the designer much greater expressive power than previous layout description methods. In this chapter I will describe in general terms some of the features of the new language, Lava, to motivate the work described in the next chapter. After describing some of the goals and major features of Lava, I will describe the primitive objects and operations of the language. Finally, I will describe a canonical intermediate form that is used as input to the algorithms of the next chapter.

3.1. Goals and Specifications for an IC Description Method

This section describes the motivation behind the design of the IC layout language Lava. I also will list some general goals for IC descriptions, and then give a set of design specifications for an IC layout language, while relating them to the goals.

Modern programming languages embody solutions to problems of managing complexity and representing structure. Similar problems exist for the task of describing hardware. In the design of Lava analogies with programming languages have proved very useful and were frequently used as a guide.

Goals

A high level of abstraction is one way to provide the designer with some "leverage" over the design. By using automation to avoid details, the designer can potentially gain much more leverage than if he had to work with the details, even with machine assistance. Potential benefits include reduced need for design auditing tools, reduced errors, and relative design-rule independence of the design. The programming language analogue would be high-level programming languages.

At odds with the goal of a high level of abstraction is that of providing a generality of application. Many special purpose layout tools have been constructed that are very powerful, but good only for a narrow range of applications. In contrast, my objective is a tool that could replace

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

current IC layout tools, produce any topology that the designer intends, and would not restrict the design methodologies available to the designer.

Also conflicting with the goal of a high level of abstraction are the designer's goals of area efficiency, speed, *etc.* The design tool must have an acceptably small performance penalty for its use.

The IC description should be concise in order to be easier to read, write, and modify.

Design specifications for an IC description method

Given these general goals, there are many ways to design an IC description method. The following are the important features of Lava.

Hierarchy has long been used to deal with the complexity of programs, ICs, *etc.* because it reduces the quantity of information that must be manipulated at any given time. As ICs are organized and designed in a hierarchical fashion, an IC description method should be capable of representing this structure. A hierarchical description also reduces the effort required for programs that implement the design description.

The notion of parameterized cells is another useful idea borrowed from programming languages. It allows variants of a cell to be created and used without the user dealing with all of the details of its construction. These parameters can be used for modifying the size of devices in a cell or they can be used to control repetition and conditional inclusion, thereby making it possible to specify a generic cell that can generate variants of a cell. Typical uses might be for specifying the width of a register, or for selecting one of a number of variant cell types, such as the least significant bit versus a center bit of an adder. Parameters lead to more concise descriptions and keep the designer from having to respecify nearly identical objects. The elimination of duplicate similar cells makes bug fixes much simpler — one only needs to fix the generator rather than all the versions of the cells.

Another simplifying concept is the use of wire aggregates. Arrays and records are constructs in programming languages for grouping together related variables and manipulating them

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as one entity. A similar concept can be very useful for describing hardware. Busses may be described as an array of connections or wires; a collection of control lines not part of a regular array can be thought of as a "record" of wires. In Lava, these records are called *bundles*. A bus including control lines would be a bundle that includes an array. Connection between such aggregates may be made in a manner analogous to plugging together multi-pin connectors (such as the connection between a terminal and a computer), provided that the shape and size of the connectors matches. Aggregates allow an IC description to be much more concise by hiding details that are unimportant. At a high level when a connection is been made between two blocks, you know that the left signals of block A connect to the right signals of block B, but the names of the individual connections are irrelevant. After all, when you last plugged together a multi-pin connector did you think about each of the many connections being made?

In Lava, the electrical circuit is the fundamental method of description. It is specified explicitly and unambiguously using a variety of language features included solely to make the electrical specification convenient and concise. Since the electrical circuit is one of the levels of specifications that the designer works through on his way to designing the IC, he should specify the electrical description directly rather than indirectly through a topological or geometric description.

In a utopian world, the electrical specification is all that would be necessary to produce an optimal layout. In reality, a better layout can be produced if the designer provides topological or geometric information such as the orientation or relative position of a component, or the width, layer, or route of a wire. Such information will be referred to as a *hint*, that is, information that aids the compiler of the description to produce a layout from an electrical description. Lava contains a number of variants on the method of description, the major difference being the types of hints required. When the electrical specification (or a higher level) is the primary specification of circuit behavior, there are fewer conversion steps to be performed by the designer, and topological and geometric *hints* need be specified only after the circuit (or chip) has been completely electrically specified and a functional simulation performed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A Lava IC description does not contain assumptions about the geometric design rules. Assumptions about the form and specifics of geometric design rules should not be a part of the description that the designer provides. The design-rule-independent description also precludes the specification of most physical dimensions, as dimensions are almost always a function of design rules. There are many benefits of a design-rule-independent description, but one of the most significant is reducing the amount of detail the designer must specify. Since the design tool now has the responsibility for maintaining design-rule correctness, it also has the power necessary to do some layout minimization and other optimizations, which will be discussed later.

A design-rule-independent description allows the design rules to be changed with little effort, thus permitting a great deal of portability of designs among fabrication facilities. Universal design-rule sets, whose aim is also to increase portability, rely on the assumption that as the technology improves, it improves uniformly for all the design rules so that designs can simply be scaled for different fabrication facilities. This assumption is poor, because different design rules have shrunk at different rates. For example, recently minimal transistor dimensions have been reduced far more than minimal line widths. While scaling still works, the scaled design gradually becomes less optimal.

Currently, design tools that offer the designer some degree of automation (*i.e.*, do more than just drafting) to aid him in IC layout often do not produce an optimal result. If the layout is not satisfactory, the designer is faced with the choice of putting up with the tool and layout or using less automatic methods. Rather than abandoning this automation, the designer should be able to provide hints, or otherwise constrain the design tool, to produce a result more to his liking. Allowing the designer to add more and more constraints results in a continuum from automatic to manual design. This continuum is important for allowing the designer to produce layouts with an acceptably small area and performance penalty.

Cell stretching and pitch matching are useful cell composition techniques for achieving efficient designs. In order to make cell stretching easy for the designer, he should not have to specify the rules by which the cell stretches. Computing the set of rules that guarantee that a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cell's geometry remains design-rule correct as it is stretched in both dimensions is a tedious and error-prone task for a designer. Computing this set of rules can better be done by machine. Automatically generated stretch rules allow the designer to work at a higher level of abstraction and make the description more concise. Because the design tool has the power and responsibility for design-rule correctness, it is capable of generating stretchable cells, as will be shown in the next chapter.

Form of the description

The previous section described some of the capabilities of the Lava language; this section will discuss some of the implementation decisions. There are a number of choices to be made when you create a new IC layout language. While I do not claim that there is a correct or best answer, I will justify the choices I have made. In general, Lava uses the form that was the most general and put the fewest restrictions on what could be represented.

The first major question was whether to use a graphical input form or a textual language as the basis for Lava. A textual description was chosen for several reasons. First, there are many concepts that are easy to express textually, but are very difficult to express graphically. For example, the ideas of repeating an object N times, where N is a parameter yet to be specified, or of conditional inclusion, where different objects are placed in the layout depending on a parameter, are more naturally expressed textually. However, graphics does seems better suited for expressing specific instances of the more general specification.

Probably more important, as long as the textual description is at least as general as the graphical description, a preprocessor can be written to manipulate the latter to produce the former. In fact, such a program has been written (called SEDIT [Burns 82]) to manipulate graphically portions of layout descriptions and produce a textual equivalent. It has proved very useful, but its weaknesses are just where one might expect: the concepts for which the graphical description does not have sufficient expressive power, such as iteration, wire aggregates, and conditional inclusion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Finally, developing a graphical description entails spending a large amount of time developing a graphical user interface. Eventually, graphics will be important in the overall solution to the problems of IC layout, but it is not essential for exploring what fundamental operations and facilities are needed and how to implement them. Therefore, the design of a graphical user interface was deferred.

Another major design decision was whether to make Lava a stand-alone or an embedded layout language. Lava is a stand-alone language, not an extension of an existing programming language. The purpose of this project was to try to understand what facilities are necessary to specify ICs, and a stand-alone language is superior for that purpose. If some feature is missing, the IC designer cannot fall back on the surrounding programming language, but must complain to the language designer. The problems become obvious this way. Many embedded design languages are merely facilities for writing design tools, but not design tools. The design decision is: *What is the language with which the designer describes his design?* not *What is the language with which to write a design tool?* Creating a stand-alone language is not giving up anything, as the stand-alone language can later be embedded in a conventional programming language by including statements or procedure calls to generate the stand-alone language statements.

The Lava input is a concise, powerful notation for hierarchical, design-rule-independent, textual IC description. Major features include an electrically explicit, but geometrically ambiguous, description supplemented with *hints* to provide a continuum from automatic to manual translation.

3.2. Lava Overview

Organization

In Lava, an IC layout is described as a hierarchy of cells. A cell may recursively contain subcells, and appear multiple times in a layout, but may not intersect another cell. A cell that contains no other cells is called a *leaf* or *leaf-level* cell. From an electrical view, cells and subcells could be called circuits and subcircuits. A cell may have a number of connections points

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

called *terminals*. A logical grouping (by means of bundles or arrays) of one or more terminals is called a *pin*. Physically, each cell has an inviolate rectangular boundary enclosing transistors, wires, vias, subcells, etc. The terminals appear only on the boundary. A pin is a logical entity and has no physical existence independent of the terminals of which it is composed.

Cells in Lava may have *parameters*. A parameter is numeric information provided to a cell by the cell above it (its parent in the hierarchy) in order to customize the cell or its subcells. The customization is done by controlling repetition, subscript indices, or conditional inclusion. There is no information returned (passed up the hierarchy) that is under control of the designer.

Features common to most cell types

Most cell types are similar except for the hints. What follows is a description of those features in common among most all cell types. I will discuss *objects*, their interconnections, and then a few points of notation.

Objects

Cells are composed of interconnected objects. There are two kinds of objects: *primitive components* (such as transistors, loads, or vias), which are defined in a technology database, and cell calls (the creation of an instance of another cell). Depending on the component, there may be optional information included: a sub-type (such as transistor type), parameters (such as dimensions or aspect ratio), transformations (such as rotations and reflections), and various hints as required by the cell type. Instances of objects may be given names, which are used to refer to that object instance.

Cell calls can either be single cell calls or iterated cell calls. A single cell call causes an instance of a subcell to be included in a cell, gives local names to each pin (group of terminals) of the subcell, and provides values to the parameters of the subcell (from expressions in the cell call). Transformations and hints may be included, as with primitive components.

An iterated cell call additionally includes a count, a direction, and a description of how adjacent faces are to be connected, and produces a linear array of identical cells. (The iterated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

call is not the only way to generated an array of cells. The `foreach` construct (see section on notation below) may also be used to generate multiple single cells calls.

The wiring operator and wire expressions

Lava is an expression-based language where the existence of an expression causes an object to appear or a connection to be made. Central to wire expressions is the *wiring operator*.

Lava's method of description is primarily electrical; consequently, interconnections are specified electrically, while the geometric or topological aspects are derived from the electrical description as much as possible. Hints are used to supply physical information that cannot be determined from the electrical description. The interconnections are specified by a symbolic equivalencing of names or expressions that represent nets or connection points. The expression might refer to a particular connection point on a primitive component or a pin on the cell boundary. An electrical description that is independent of physical information allows the circuit to be designed and its description to be debugged prior to layout (by means of functional simulators and circuit-analysis tools). The equivalencing is done by an operator "#" which will be referred to as the *wiring operator*, but it is important to remember that the operation actually being performed is to make its operands electrically equivalent, and may or may not cause a wire to be generated.

There are several types of permissible operands for the wiring operator. For one, a pin of the cell being constructed may be equivalenced. These pins are known as *formal pins* in analogy to formal parameters of programming languages. Another is a terminal of a primitive component. Such a terminal is just a reference to the component, possibly with a qualification to specify which terminal if the component has multiple terminals. An entire wire expression may itself be used as an operand in another wire expression. Lastly, an otherwise unknown identifier is simply equivalenced to the other operand and may then be used elsewhere in the cell to represent those net(s). What about connections to the pins of subcells? Subcell calls have implied wiring operations for each of the pins. Thus, an identifier used as an *actual pin* name (analogous to an actual parameter in programming languages) becomes synonymous with the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subcell pin. This same name may be used in another subcell call, making a connection between the cells without introducing additional names or statements. More on this technique appears in the section on notation.

In addition to wiring together individual nets, the wiring operator is capable of wiring together corresponding elements from aggregates of nets. These aggregates are the hardware designer's equivalent of the programmer's arrays and records. In Lava they are called arrays and bundles, respectively. Roughly, an *array* is a collection of connections or nets where the individual elements are of similar structure and are accessed by a numerical index, whereas a *bundle* is a collection of inhomogeneous connections or nets accessed by element name. Either a bundle or an array may have bundles or arrays as elements.

To prevent unintentional or nonsensical connections, the wiring operator only permits wiring together things of identical structure. If the operands are arrays, they must be of the same size and the elements must be of the same structure; if the operands are bundles, there must be the same number of elements, and the corresponding elements must be of the same structure. Essentially, if we are looking at the trees corresponding to the aggregate's structure, the trees must have the same size and shape.

An aggregate's structure may be defined in several ways. First, pins of cells may be explicitly given structure in the pin declarations. Second, the structure of terminals on primitive objects is known (scalar) and the structure of pins on subcells is known (from the subcell pin declarations). Third, wiring to something of known structure will propagate the structure information to an identifier of unknown structure. Since subcell calls have an implied wiring operation, they are just a special case of this structure propagation. Thus, an element selected from an aggregate also has known structure, and therefore may be used to define the structure of something else. There are some examples of this kind of structure inference in the next section. Finally, an iterated cell call creates arrays whose elements have the structure of (and are wired to) the pins of the subcells.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Notation

A design in lava consists of a set of bundle definitions and cell definitions. The bundle definitions give a name to a bundle type so that pins throughout the chip can be conveniently given the same structure. A cell definition consists of a cell header, pin declarations, and the cell body. The cell header and pin declarations name the cell, the pins, and the parameters, and specify properties (including the structure) of pins.

The cell body is a block of statements. In Lava, the order of statements within a block is irrelevant, as they are not executed sequentially (or even executed) as they would be in a programming language.

There are two statements for conditional inclusion and repetition: `if` and `foreach`. The `if` statement chooses one of two blocks of statements based on the value of an arithmetic expression, which may be a function of parameters. The `foreach` statement causes a block of statements to be reproduced some number of times, possibly with a variation. The number of time the block is reproduced can be a function of the cell's parameters. An index variable is set to a different integer for each copy of the block of statements. It can be used within the block in arithmetic expressions just as cell parameters are used. These arithmetic expressions can be used as parameters in subcell calls, to control `if` or `foreach` statements, *etc.* For example:

```
foreach x to 5 {
    if(x < 3 || x > 4) {
        cell_call(pin; x+j);
    } else {
        other_cell_call(pin2; x);
    }
}
```

is equivalent to:

```
cell_call(pin; 1+j);
cell_call(pin; 2+j);
other_cell_call(pin2; 3);
other_cell_call(pin2; 4);
cell_call(pin; 5+j);
```

Lava is an expression-based language, and the remaining statements are of one of two types of expressions: instance expressions or wire expressions. As with operators in arithmetic

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expressions, Lava's instance or wire operators have precedence and associativity, which may be modified by parentheses. What is different is that the expressions do not have values, but express relationships. Most binary operators say something about the relationship of things referred to by their operands. If an expression is being used as an operand, it refers to one, or the other, or some combination of its operands.

In general, instance expression cause objects to exist, and describes physical (geometric or topological) relations among them, while wire expressions denote electrical relationships between these objects (connectivity).

The simplest examples of instance expressions are cell calls and declarations of primitive objects (such as transistors). There are operators such as `above` or "`|`" (abuts) that express physical relationships. The objects referred to in an instance expression can be given a name, which can be used to refer to them elsewhere. This name is called an *instance name*. For example:

```
name = instance-expression
```

This instance name may be used wherever an instance expression is required. Instance names may also be subscripted:

```
insname[5] = instance-expression
```

which is useful for working with iterated structures.

Simple examples of wire expressions are pin names (from the cell header), or actual pin names (from a subcell call). The most important wire expression operator is the wiring or equivalencing operator "#". It connects or equivalences the nets of its operands. When used within another wire expression, the nets that are being referred to (*i.e.* its "value") are simply those of one of its operands[1]. If the wire expression refers to an array or bundle, subscripting or qualification, may be used to select parts of the aggregate. This selection is done with square brackets "`[ ]`" or dot "." as in many programming languages. To connect to a primitive

[1] Which one only matters in cell types where "#" produces wire segments rather than net lists. In such cases the "value" is the right operand, and "#" is left associative. This convention is arbitrary but will cause the sensible result for expressions such as `a # b # c`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

component, an instance expression referring to a single component may be used as a wire expression if the component has a single net (a # via br) or with qualification if a multiple-terminal component (a # tranname.gate). The following sample wire expression illustrates some of the above features:

```
((busa # busb).data # aluin)[1] # highbit
```

Where busa and busb are bundles that have an element data which is an array of the same size as aluin.

Cell types and the hints they require

The electrical circuit alone is not enough for the designer to specify the layout that he desires to produce. Lava's compilation algorithms require an initial, almost correct, though not necessarily optimal, layout. The information from which this layout is determined varies according to cell type. Additional information such as the relative location of components, the layer or width of interconnect, *etc.*, may be necessary. These hints will be described in this section for each of the three cell types: Stix, Abut, and Externals cells.

The purpose of the Stix cell type is for designing lower level cells where the designer has a specific idea of the layout he desires, and wishes to have a high degree of control over the locations of components and wires. In this cell type, relative positions are specified by giving each of the components *pseudo-coordinates*. These pseudo-coordinates merely establish a relative position; the magnitude has no meaning except in comparison with other coordinates. For example, the coordinates from a stick diagram would serve well as pseudo-coordinates. Because subcells may have several connections on a side and are potentially stretchable, rather than a pseudo-coordinate representing the location of its center as with other components, the subcell requires pseudo-coordinates representing each of the edges[2].

In Stix cells, wires generate single wire segments with the color (layer), width, direction and pseudo-coordinate provided as hints. In many cases the color, pseudo-coordinate, and

[2] Provided as a pseudo-coordinate representing the lower left corner and a *pseudo-size*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

direction may be inferred from the adjacent components. The width will default to the minimum width.

Other hints that may be specified are transformations (rotations and reflections) and user constraints. User constraints are constraints on the relative position of objects. These differ from pseudo-coordinates in that pseudo-coordinates only provide an initial relative position of objects, which may change during compaction, whereas constraints always remain in effect.

The need for pseudo-coordinates in the Stix cell type makes the use of wire aggregates (bundles and arrays) sufficiently awkward to be not very useful. A distinct pseudo-coordinate needs to be given to each element of the aggregate. When several elements of a bundle or array are connected to the same subcell, pseudo-coordinates cannot be inferred and must be provided manually.

The Abut cell type is intended for the design of larger cells that are composed entirely of subcells. The principle distinction between Abut and Stix cell types is the manner in which the initial positions are specified and a restriction on the components allowed (only subcells in Abut).

In the Abut cell type, the area of the cell is entirely tiled by the subcells and all connections are made by abutment. The initial positions are specified by the operators such as `left of`, which specifies that a cell or group of cells is to be to the left of another cell or group of cells. Another set of operators, of which `abuts left of` is an example, specifies that connections between the cells (or groups of cells) are to be made by abutting and that the cells are to be adjacent[3]. Because of the tiling, no pseudo-coordinate, pseudo-size, or wire direction need be specified. Also, because the only components are subcells, the wire layer can always be determined from the subcells. Wire widths and transformations are hints that may still be specified.

The External cell type provides a way to use cells that were designed outside of Lava. This cell type simply provides a mapping from cell and pin names to CIF cell number, cell size, and

[3] The distinction between `abuts left of` and `left of` would seem redundant. The intention was that connections that were not made by abutment would be saved for a wire router. Unfortunately, this feature never got implemented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pin location. One could view the hints for this cell type as being the complete, rigid layout.

Why different cell types? At each level in the cell hierarchy cells are composed of a number of component objects. The kinds of information that the designer would like to and not like to specify are different at the different levels. At the bottom level (*leaf level*), the objects being composed are typically single devices, and the connections are single wire segments. The relative locations are specified for most all components. In top level cells, the objects being composed are often themselves large cells with thousands of devices in them. The interconnections consist of hundreds of individual wires, which are often placed in logical groupings and wired systematically. These wires are often not just straight wire segments, but are routed around objects and change layers automatically. The relative placement of objects are more loosely specified. In general, Lava is given fewer hints and more freedom in the specification of larger cells because the optimum position, route, or stretch is frequently not known by the designer. At lower levels, the designer often wants more exacting control over the operations performed by the design tool and so the hints should be capable of fine control. At the higher levels, such fine control is not necessary. The designer often wishes to use broad strokes to describe layout at this level.

3.3. Intermediate Form

Lava undergoes a partial compilation to remove parameters, conditionals, iterations, bundles, arrays, *etc*. This expanded form is the input to the symbolic layout algorithms of the next chapter. Since at this level the description is no longer parameterized, a cell name consists of both the name given by the designer plus the list of parameter values that were used to produce this "distilled" description. The contents of the expanded version of two different cell types, Stix and Abut will be described.

For both cell types, the majority of the information in the intermediate form is contained in a component list and a wire list. Components are terminals, subcells, transistors, vias, *etc*., although in the Abut cell type only the first two are permissible. Other information in the component list, which is not specific to cell type, is additional information about the type of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

component, parameters (such as sizes), orientation, *etc.* The connection points on the components are connected together only by wires, which are horizontal or vertical elastic segments.

In addition to the components and wires, the algorithms of the next chapter sometimes need to know connectivity information for this cell. Such information can be computed from the component and wire lists, and from knowledge of the connectivity of the components. The external connectivity information (called the *electrical abstraction*) for the subcells was computed during a similar prior operation. From the connectivity, an electrical abstraction for the current cell is prepared for use by its superior cells.

In the Stix cell type, an initial (pre-compaction) position for all of the components and wires is necessary to correctly generate the compaction constraints. As these initial positions are only used to determine an ordering, and have no relationship to any physical dimensions, they are called *pseudo-coordinates*. Also, if a subcell has more than one connection on a side, the pseudo-coordinates for all sides of the cell are necessary[4]. Wires have a single pseudo-coordinate and a direction. These pseudo-coordinates may be thought of as coordinates in a picture of the stick diagram of the cell.

In practice, much of the information such as positions, direction, layer, *etc.* may be inferred from positions, layer, *etc.* of adjacent components and wires. Lava does this inference so that the user does not need to provide much of this information. For the purposes of the discussion it will be assumed that all that information has already been filled in.

In the Abut cell type, the designer provides the information with which to construct an initial placement in a different manner. There are no pseudo-coordinates in the component or wire lists. The only additional information is a list of instance expressions giving the relative locations of subcells or set of subcells (such as `A leftof B`) as discussed in a previous section.

These lists of components and wires constitute most of the essential information required by the constraint-generation algorithms of the next chapter. The issues involved in producing these

[4] These pseudo-coordinates are provided as a pseudo-coordinate pair and a pseudo-size pair.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

lists from the Lava input are mostly language processing and compiler issues and are not particularly relevant to IC design tools. A different front end (such as a graphics interface) could easily be substituted at this point and produce the same lists as output.

Summary

In this chapter, I discussed some of the goals of producing a layout tool that allowed the designer to represent his design at a high level of abstraction with minimal sacrifice of flexibility or silicon area. I gave an overview of the language, which describes electrical connectivity, showing some of the major features: wire aggregates, parameterizable cells, an overly flexible description with disambiguating hints, and a design-rule-independent description. Finally, I described the intermediate form which is the input to the algorithms of the next chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Lava Compilation

The previous chapter discussed methods to describe IC layouts using a language called Lava. This chapter presents methods for compiling Lava descriptions into a few well-defined optimization problems. The resulting optimization problems will be relatively small and sparse, and can be solved by methods described in the following chapter.

The compilation of the Lava description can be converted into three different optimization problems, *longest path*, *L^1 minimization*, and *constraint propagation*. These optimization problems will be defined precisely as they are encountered in this chapter.

A second purpose to this chapter is to set the stage for the next chapter in which the characteristics of the resulting optimization problems are studied. In order for the characterization to be meaningful, the precise manner in which the problems are constructed must be specified. The above, however, does not mean that the characterizations will not be typical of optimization problems created by other symbolic IC layout systems. I expect the characterizations to be typical of other symbolic layout systems as well.

4.1. Sticks

In this section I will first talk about standard sticks compaction, then I will extend it to take advantage of the existing hierarchy of the chip, and discuss some extensions that improve the quality of the layout and the speed of compilation. Finally, I will summarize with a perspective that unifies all the optimization problems used in this section.

The object of sticks compaction is to produce a minimal cell size for a given circuit while obeying all the constraints imposed by the design rules. The optimal solution (minimal area) of such a problem is computationally very difficult[1]. However, the problem can be changed to make it easier in several ways. First, the designer is often willing to (or demands to) provide the orientation and an initial relative placement of each component. The circuit description with this

[1] To optimally solve such a problem, all legal combinations of relative positions, rotations, and reflections must be considered to optimize the nonlinear objective function. Optimal sticks compaction has been shown to be NP-complete [Sastry 82].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information is similar to a stick diagram in information content. While the size of the solution space has been cut tremendously, finding a solution of minimal area is still a difficult problem. The objective function is nonlinear and the constraints are also nonlinear (conditional), making the solution space nonconvex.

If the problem is broken down into a sequence of one-dimensional problems (alternately solving for minimum width and height), the result may not quite be optimal, but the optimization problems become very tractable. The objective (minimizing only the height or width) is linear, and the constraints are linear, resulting in a convex solution space. This problem is the longest-path problem, which is a special case of the linear programming problem. The next section describes the basic compaction algorithm for sticks-like descriptions. This method will be extended to gain efficiency (by using hierarchy) and quality (by using L^1 minimization) in the following sections.

4.1.1. Standard Sticks Compaction

There are four types of constraints used in compaction. The first three types: *component constraints*, *connection constraints*, and *boundary constraints* are called *fixed constraints* because they are computed once and do not change from one compaction step to another. The fourth type, *design-rule spacing constraints* are called *variable constraints* because they are a function of the current positions of components and therefore may vary from one compaction step to the next. These four constraints are used to determine the positions of the objects in the final layout.

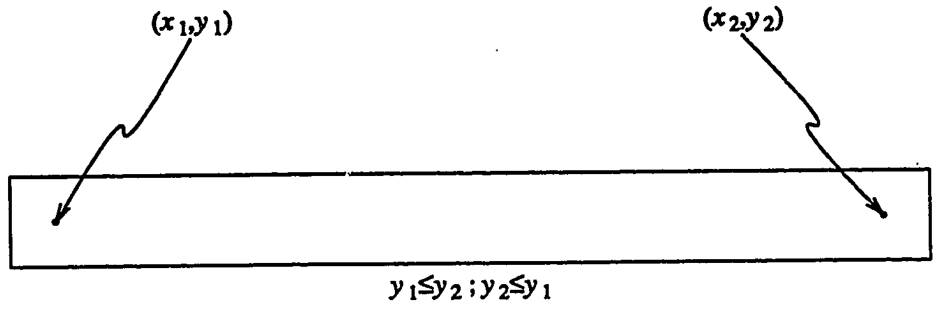


Figure 4.1 The Component Constraints for a Wire

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Component constraints are used to hold portions of each component together in some specific relationship. Many component are not flexible or stretchable and therefore have no need for such constraints. An example of a simple component that requires these constraints is a wire. In Figure 4.1, the constraints $y_1 \le y_2$ and $y_2 \le y_1$ keep the horizontal wire horizontal, while remaining elastic in the x direction. A more complex component example is a depletion load with butting contact (pullup) and is shown in Figure 4.2. Here, in this depletion load, the butting contact

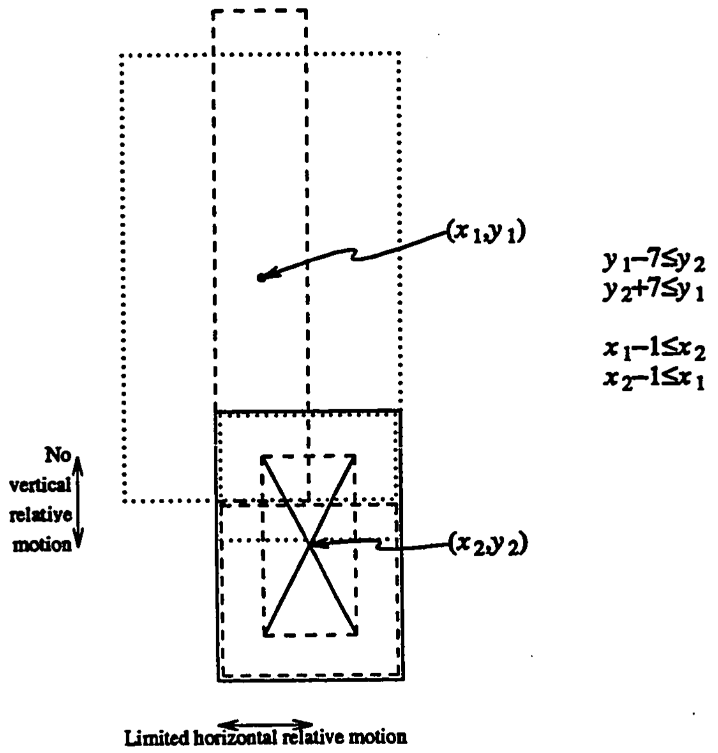
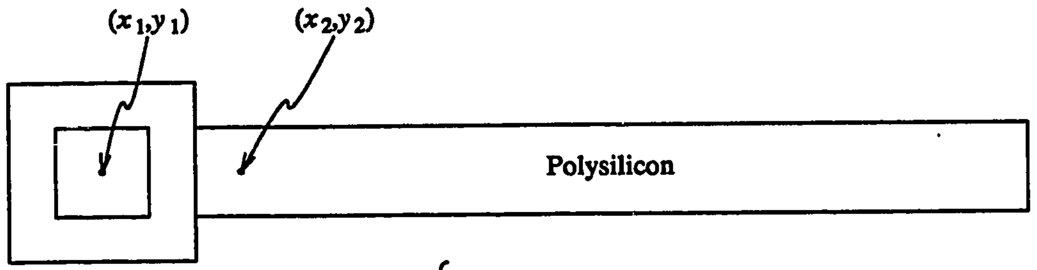


Figure 4.2 The component constraints for a depletion load

has a limited range of motion with respect to the transistor and this range is set by the component constraints. Component constraints are very useful for hierarchical sticks, since the subcells become components, and the component constraints represent the pin constraints of the subcell. The next section discusses this in more detail.

Connection constraints are used to hold together two connected components, and to maintain the design-rule-specified minimum width for the connection. In Figure 4.3 a polysilicon wire is connected to a poly-to-metal contact. The constraints in x hold the connection together, while

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

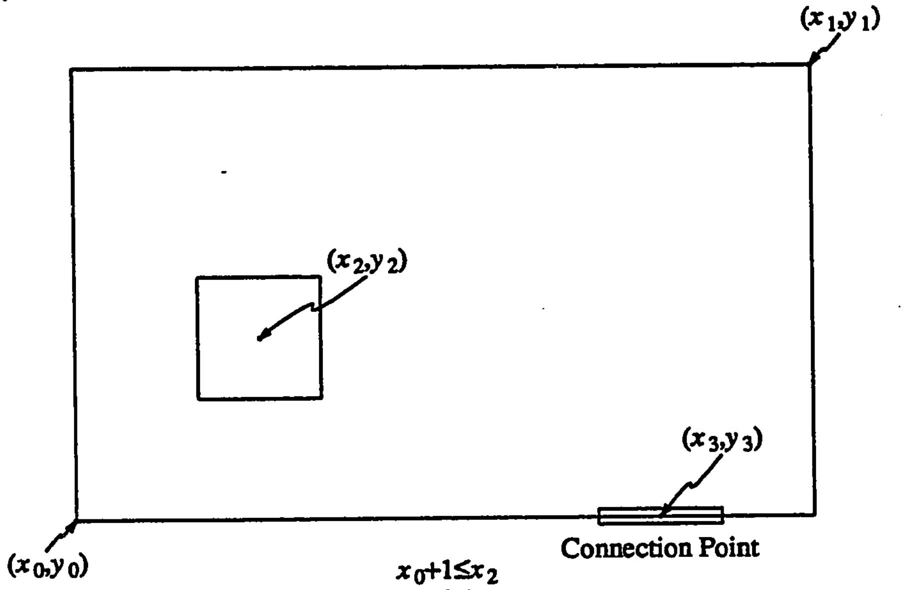


$$x_1+3=x_2 \quad \begin{cases} x_1+3\le x_2 \\ x_2-3\le x_1 \end{cases} \qquad \begin{matrix} y_1-1\le y_2 \\ y_2-1\le y_1 \end{matrix}$$

Figure 4.3 The Connection Constraints for a Wire and a Contact

the constraints in y ensure at least a minimum width connection. Note that the wire is not restricted to connect on center, but may slide one unit either way.

Boundary constraints simply constrain all components to remain within the four boundaries of the cell. Also, connection points (terminals) are constrained to lie on the boundary. See Figure 4.4.



$$\begin{matrix} x_0+1\le x_2 \\ x_2+1\le x_1 \\ y_0+1\le y_2 \\ y_2+1\le y_2 \end{matrix}$$

$$y_3=y_0 \quad \begin{cases} y_3\le y_0 \\ y_0\le y_3 \end{cases}$$

Figure 4.4 Boundary Constraints

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Design-rule spacing constraints, rather than holding the components together, keep them apart. As a consequence of decomposition into separate x and y problems these constraints must be a function of the positions of components[2]. For the sake of this discussion assume that constraints are being generated for compaction in the x direction. Each pair of components is examined to determine if design-rule spacing constraints should be written between them: If their spacing in y is less than the design rules between these components, then a constraint is generated, keeping the components greater than this design rule distance apart[3]. If the pair of components are within this distance in y, then an x constraint must be generated between these objects. The current x positions are examined to determine the direction of this constraint, *i.e.*, is A constrained to be to the left or right of B?. The appropriate constraint is generated as in Figure 4.5.

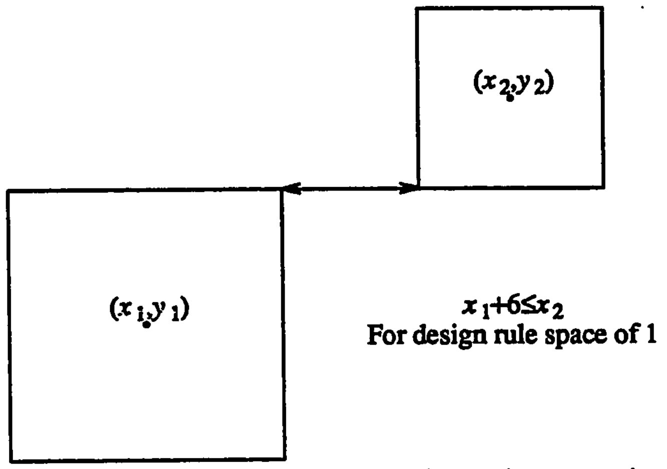


Figure 4.5 The generation of a design-rule-spacing constraint

Having constructed a graph representing all of the constraints between the components, we can find the positions of the objects by solving a standard optimization problem — finding the longest path in a graph. The longest-path problem is defined as follows:

[2] Actually, by generating many additional constraints, the design rule constraints can be made independent of the positions of the components; however, these additional constraints tend to bind and overconstrain the cell, preventing much useful compaction from occurring.

[3] The design rule distance is a function of layers and electrical connectivity.

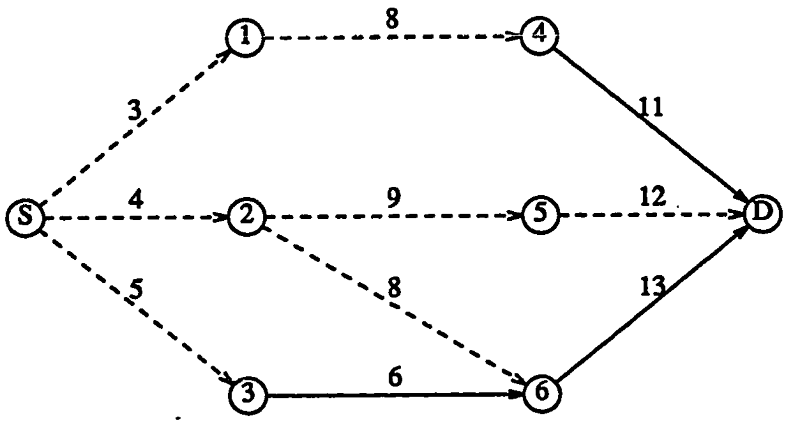
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Given a set of constraints

$$\left\{x_i + c_{ij} \le x_j\right\},$$

where the c's are unrestricted in sign,
find a solution such that $x_K - x_L$ is minimized for a given K and L.

This problem can also be represented by a directed graph, where the x's are vertices, $x_i + c_{ij} \le x_j$ is an arc of length c_{ij} from vertex i to vertex j, and the longest path from vertex L to K is sought[4]. The problem is isomorphic to a shortest-path problem with the opposite signs on the c_{ij}'s. There are many ways to solve this problem, some of which will be discussed in the next chapter.



dashed lines (- - - - ->) are critical paths

Figure 4.6a Graph with critical paths

Longest paths from S to vertex:	path length
1	3
2	4
3	5
4	11
5	13
6	12
D	25

Figure 4.6b Longest paths of graph in figure 4.6a

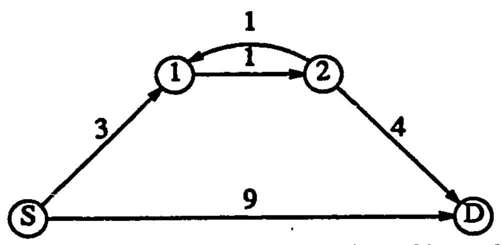
Typically there will be a single source vertex (usually representing an edge of the cell) from which the length of the longest path to each of the other vertices is desired. Figure 4.6 shows a graph with source vertex and longest paths marked. Some observations are in order. For there to

[4] For sticks compaction, vertices L and K usually represent the edges of the cell.

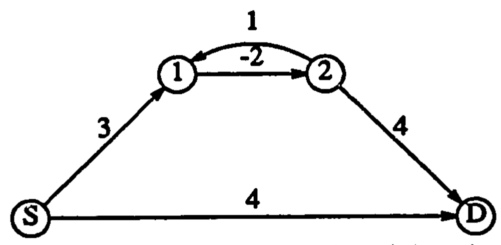
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be a finite solution to the longest-path problem, there must be at least one path from the source to the destination vertex, and none of these paths may be incident to a closed path (circuit) whose sum of arc length is positive (for the graphs we will encounter, this condition means no positive weight circuit at all in the graph). Note that closed loops with negative total lengths are permitted (and will be used frequently). Positive length loops imply a system of inequalities with no solution. Figure 4.7 shows examples of these three conditions and explains them.

a) No Path: No Longest Path. Length of longest path is $-\infty$



b) Longest Path is Infinite: It includes an infinite number of loops between nodes 1 and 2.



c) Longest Path is 5. Negative length loop is ok.

Figure 4.7 Three interesting types of constraint graphs

To perform a compactions all the fixed constraints are generated, then the variable constraints for the first compaction direction are generated (say x). The longest paths from the vertex representing the left edge to all the other vertices are computed. These distances are the new x positions of all the components (the left edge is at x=0)[5]. The variable x constraints are now

[5] Compaction to the right can be performed by modifying this procedure slightly: the distances from every vertex to the right-edge vertex are computed, and, to keep the cell in the first quadrant, subtracted from the distance from the left to the right edge.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

discarded and the variable y constraints are computed to compact in the y direction in a similar manner. Typically, only small improvements are made on the second compactions in each direction, and rarely are improvements made on the third.

Generating an Initial Feasible Solution

In order to separate the compaction problem into two one dimensional problems, we assumed that we had a set of initial coordinates for each component that described a design-rule-correct layout. Where did this set of coordinates come from? The user could have provided a consistent set of initial positions. For the most part, providing the initial positions would be easy: The positions from the stick diagram could simply be scaled up sufficiently so that all components are sufficiently far from other components. These stick-diagram positions are called pseudo-coordinates and are defined as only having a meaning in a relative sense: they only establish an ordering and do not have a distance metric.

Why is scaling necessary? How big of a scale factor should be used? The purpose of the scaling factor is to cause the distance between components to be greater than the size of any component. This scaling of the stick diagram coordinates would produce an initial, consistent, design-rule legal layout, except for one problem. The connection to some components, such as butting contacts and depletion loads, must be off-center as in Figure 4.8. Because of this off-

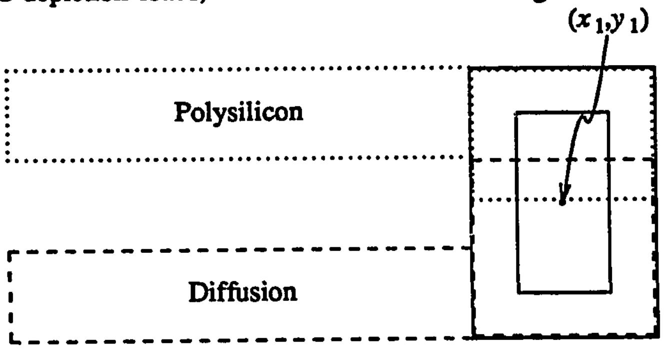


Figure 4.8 Off-center connection to butting contact

center connection, scaling will not produce an initial correct layout, as a correct layout requires a small fixed offset from the pseudo-coordinate derived position.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To derive an initial legal position, a special compaction step is used. Instead of the usual variable constraints, some constraints derived from the pseudo-coordinates are added to the fixed constraints and compaction steps in x and y are performed. The constraints are of the form $x_0+PC_x \cdot S \leq x_i$, where x_0 is the left edge, PC_x is the x pseudo-coordinate for the object (or part of an object such as a wire), x_i is the variable associated with the object, and S is a scale factor chosen to be much larger than any component. Analogous constraints in y are also generated. Compaction is to the left and down. The idea behind these constraints and compactions is to keep the components apart without preventing the fixed constraints from doing their work. The result is an initial design-rule-correct layout. Figure 4.9 illustrates the transformation from pseudo-coordinates to initial positions.

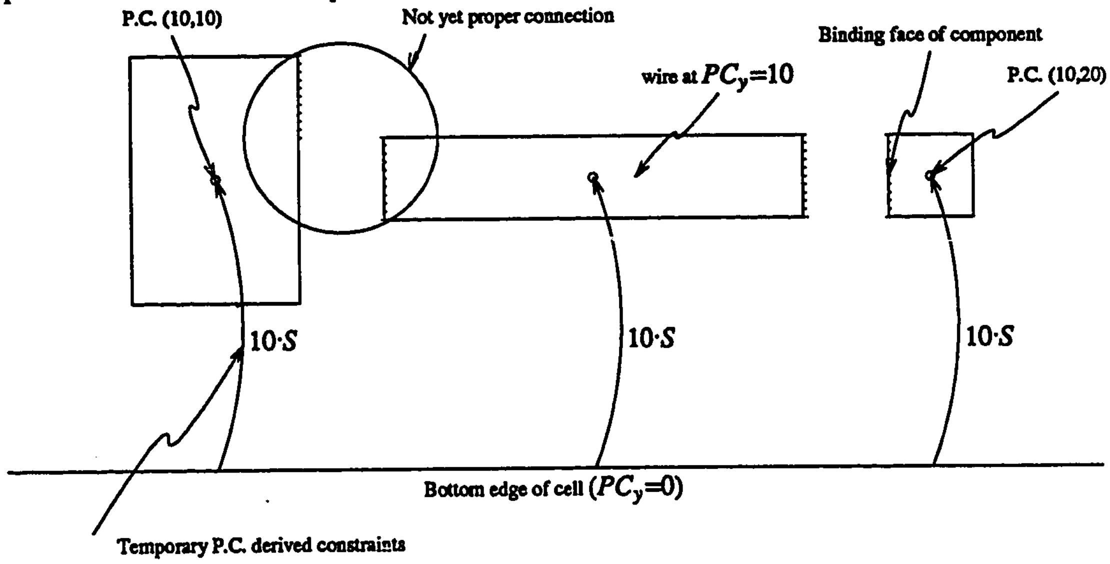

P.C. (10,10)
Not yet proper connection
Binding face of component
wire at PC_y=10
P.C. (10,20)
10·S
10·S
10·S
Bottom edge of cell (PC_y=0)
Temporary P.C. derived constraints

Figure 4.9a Before initial-feasible generation step.

Why is an initial correct layout needed? Why not just go ahead and generate variable constraints and compact? Shouldn't these off-center problems fix themselves after a few compaction steps? There are two reasons to start with an initial correct layout: The first is error diagnosis. If an initial correct layout cannot be produced, the error in the input can easily be isolated. Once the component motions have taken place, tracing the origin of a problem becomes difficult. The second is that there are cases, such as in Figure 4.10, where the pseudo-coordinates alone do not give enough information with which to choose the direction for design-rule spacing constraints,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

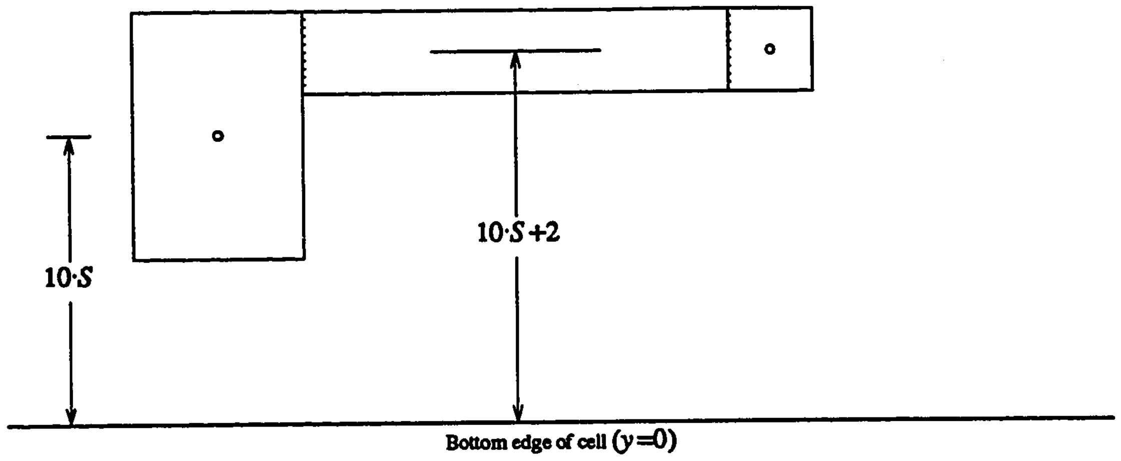
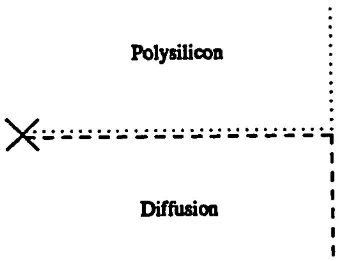
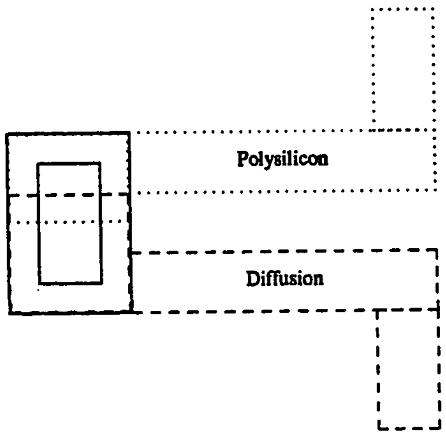


Figure 4.9b After initial-feasible generation step.

but information on the structure of components would be needed. The initial-feasible generation generation step employs this structure information.



a) Stick diagram. Contact and wires have same y pseudo-coordinate.



b) Geometry.

Figure 4.10 Ambiguity of Sticks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Why do we need pseudo-coordinates (and pseudo-sizes)? When generating spacing constraints, two questions need be answered:

a) Do we need to generate a constraint? The answer is based on the relative position in the other dimension.

b) If we generate a constraint, which way does it go? This answer is based on the relative position in the same dimension as the constraints being generated.

These pre-compaction relative positions are given by the user in terms of pseudo-coordinates. Since only relative positions are required, no assumptions are made (except for ordering) on the relationship between pseudo-coordinates and physical coordinates. In other words, there need not exist a function that will translate a pseudo-coordinate to a physical coordinate.

When working with simple non-stretchable components, the size of the component is known ahead of time and can be used in answering question a above. When stretchable subcells are used as components, pseudo-coordinates are needed for each of the edges of the subcell. In Lava, these are derived from a pseudo-coordinate that refers to the lower left corner of the subcell, and a pseudo-size. The pseudo-size need not have any relationship to the actual size (stretched or unstretched) of the cell, only to other pseudo-coordinates and pseudo-sizes. pseudo-coordinates are in essence "picture coordinates" for a sketch of the layout.

In a leaf level cell (only simple components), pseudo-coordinates were not specified for the wires because they were inferred from the adjoining components. When the components are more complicated, such as subcells, there does not exist a single pseudo-coordinate that can be assigned to the wire. In some cases, the position of the wire relative to other objects cannot be unambiguously chosen, even when taking into account the structure of the connected subcells. In Figure 4.11, without specifying a pseudo-coordinate for the blue wire, it cannot be determined whether the blue wire is above or below cell A. The structure of subcells will be further discussed in the next section.

In this section, I have outlined the algorithms for translating a stick diagram into a set of longest-path problems that will produce a maximally compacted cell, beginning with generating

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

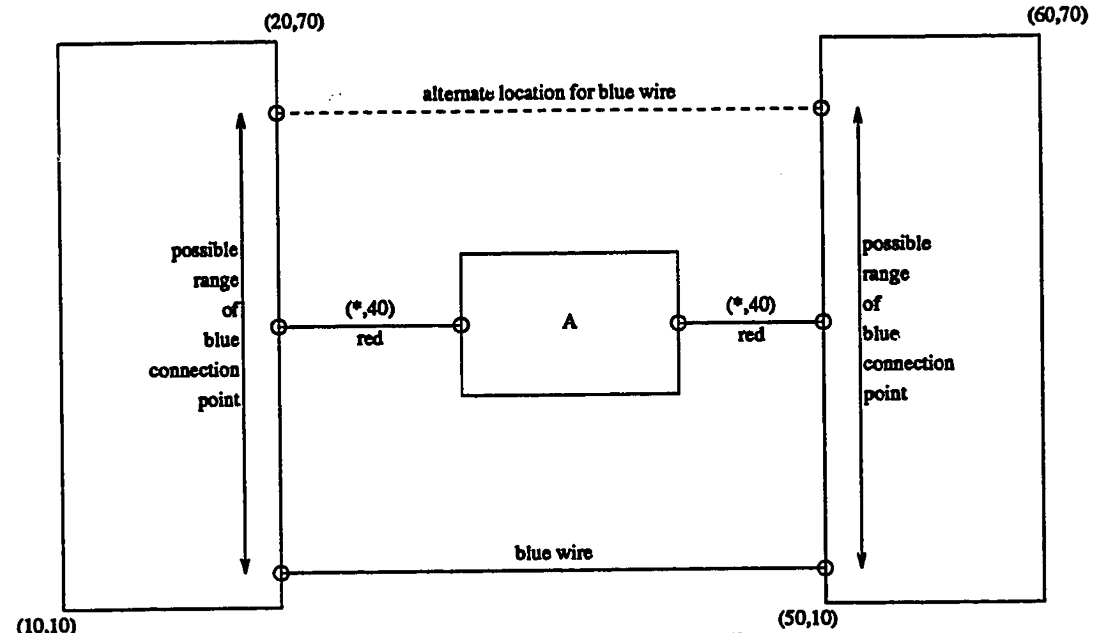

(20,70)
(60,70)
alternate location for blue wire
possible range of blue connection point
(*,40)
red
A
(*,40)
red
possible range of blue connection point
blue wire
(10,10)
(50,10)

Figure 4.11 Necessity of pseudo-coordinates

an initially correct but uncompacted layout, then adding the variable constraints and compacting one dimension at a time.

4.1.2. Hierarchical Sticks

Sticks compaction as described above is unfortunately limited in it's abilities. It cannot be used to compile an entire chip because of the unreasonably large amount of CPU time and memory required for a problem of that size. While compaction works for leaf level cells, optimally compacting these cells is frequently not the desired goal since it usually does not produce an optimal chip, as discussed in Chapter 1. Instead we need to produce a stretchable cell that can then be composed with other cells to form an efficient layout.

This section describes a method to generate a complete chip from stretchable subcells. First, I describe how to construct stretchable cells automatically. An important feature of these stretchable cells is that stretching in one dimension will neither prevent (nor restrict) stretching in the other dimension, nor cause design rule errors due to stretching in the other dimension. This decoupling of dimensions will be achieved without a significant limitation in the cell's flexibility. Next, I will show how to reduce the information in the stretchable cell description, producing an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

abstraction, to significantly reduce the CPU time required to process a hierarchical chip. Lastly, I will describe how to put these abstractions together to compile an entire chip.

Stretchable Cells

To create stretchable cells, a set of rules is needed that describe the manner in which a cell can and cannot stretch. To be useful, these rules must constitute a promise that a correct cell can be produced if the amount of stretch chosen obeys these rules. Otherwise, composition of stretchable cells would at best be a trial and error operation.

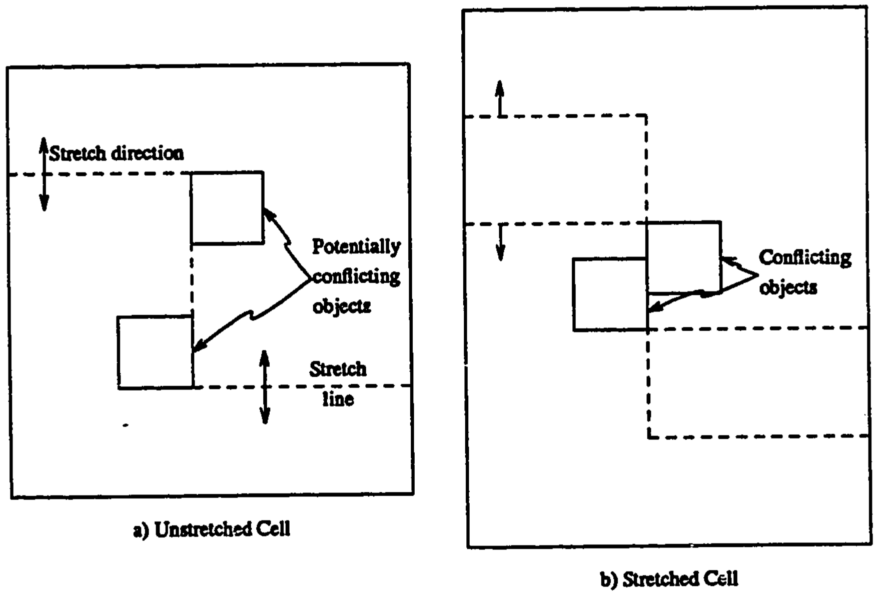
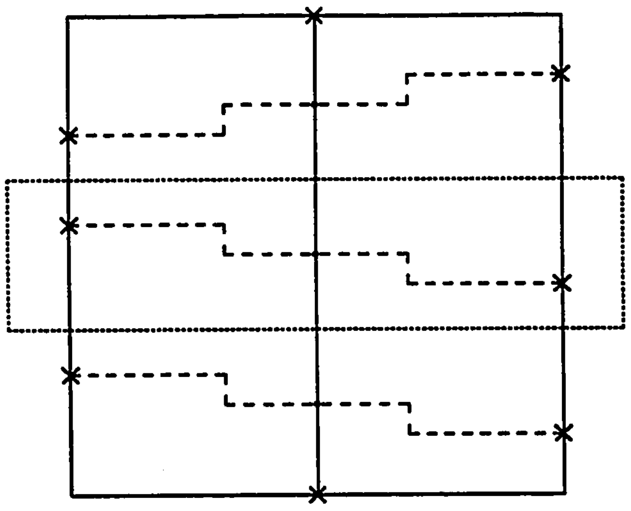


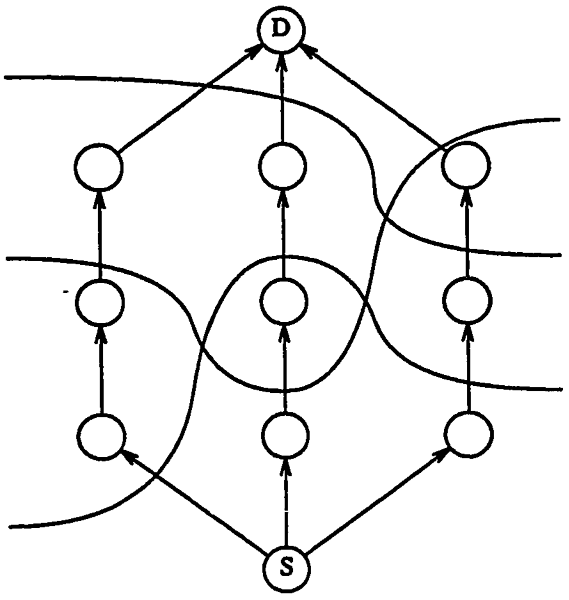
Figure 4.12 Hazards of arbitrary stretch lines

Some cell composition systems allow the user to choose *stretch lines*. However, stretching along arbitrarily chosen stretch lines will not necessarily produce a design-rule-correct layout (as in Figure 4.12). Also, describing stretch rules that do not unnecessarily limit the degree of flexibility may require describing a number of stretch lines that is worse than quadratic in the number of connection points. In Figure 4.13a, if the section of the stick diagram in the dotted box is replicated vertically n times, the resulting circuit will have a constraint graph of the form shown in Figure 4.13b. There are $(n+1)^3$ stretch lines, which correspond to edge cut-sets of the constraint graph, for this layout.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



a) stick diagram for n=3



b) Constraint graph for n=3 showing 3 of the 64 possible stretch lines (graph cut sets)

Figure 4.13 Overabundance of stretch lines

The constraints used during compaction almost constitute a completely flexible set of stretch rules. Every constraint can be traced to a design rule requiring its presence. The problem with using these constraints is that these stretch rules are actually slightly too flexible. The x and y constraints are computed based on the current values of the y and x variables, respectively. One set of constraints is identical to the one just computed in the last compaction step. Since

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

each of the constraint sets were based on the current x and y positions, we know that each set of stretch rules, by itself, is correct. However, once the cell has been stretched in one direction, say x, the x values, upon which the y constraints are based, have changed and the y stretch rules may no longer be valid. See Figure 4.14 for an example of how this may cause troubles.

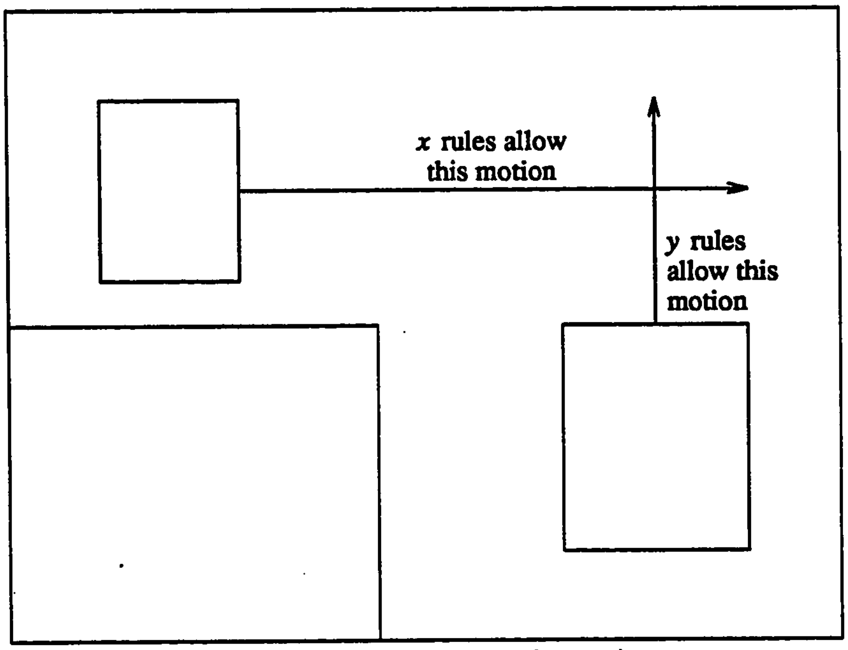


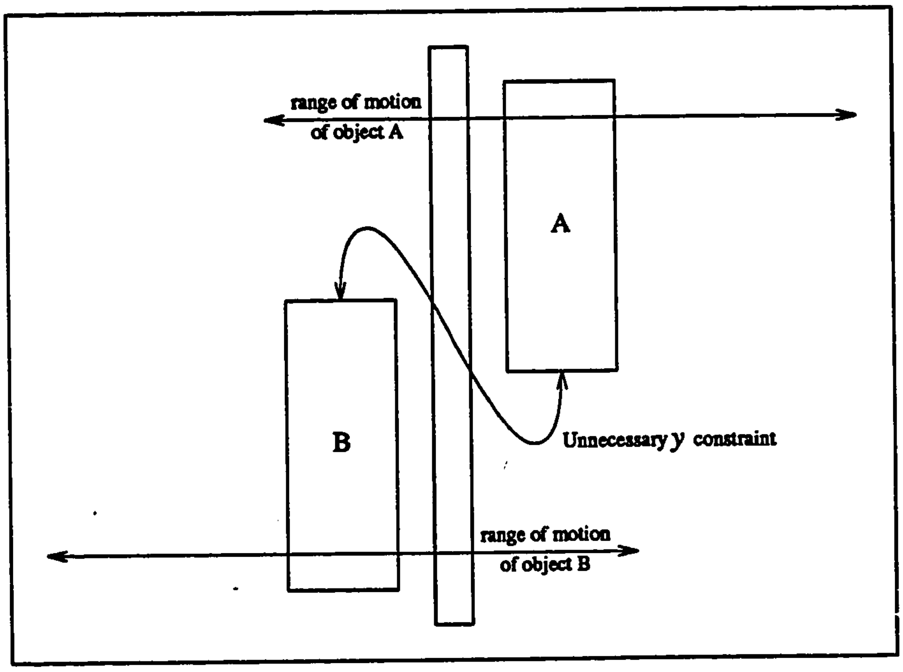
Figure 4.14 Undesired x/y interaction

This problem is called the *x/y interlock problem.* One or both of the sets of constraints need to be modified to ensure components do not interfere. I chose to modify only one set of constraints. Assume that the last compaction was performed in the x direction. The constraints from this compaction are used as the x stretch rules. A set of y constraints is constructed using a modified method of constructing the variable constraints. In the usual method of y constraint generation, the x positions of a pair of components are examined to determine if their proximity in x requires a y constraint to be written. However, when generating stretch rules, because of the possible x stretching, their x positions are not completely known. However, it is still possible to make some useful statements about their x proximity by consulting the x constraint graph.

The y constraint may be omitted only if the pair of components will *never* be close in x. To make this determination, compute the longest paths between the vertices representing the components. If one or the other of the longest paths is greater than the corresponding x

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

design-rule spacing (these spacings may be different if the components are asymmetric) then the components will never be close. Computing the y stretch rules then involves computing longest paths between many (almost all) pairs of vertices. Computing all pairs of longest paths is the same as computing a closure on the constraints graph and can be done in better than $O(v^3)$ time[6].



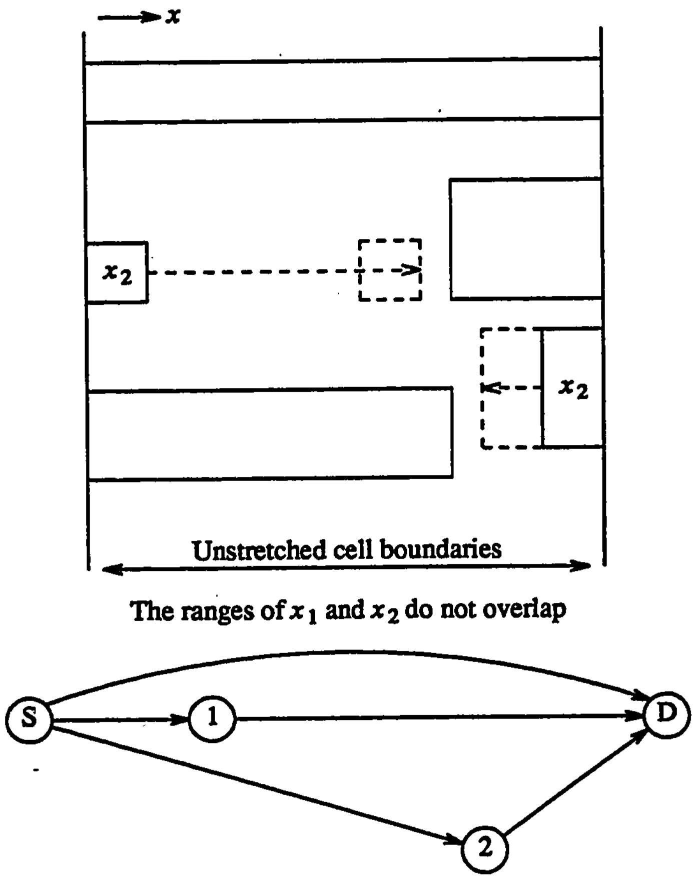
Ranges of x motion overlap resulting in an unnecessary y constraint
Figure 4.15 Additional y constraint

Dan Perkins proposed a modification to this method of computing stretch rules that is faster, but produces a somewhat more rigid set of stretch rules. Rather than computing a pair of longest paths for each pair of components, the sets of values from a compaction to the left and a compaction to the right are computed. This procedure computes a range of motion (in x) for each individual component. These ranges are compared and y constraints are generated if they overlap. Since computing these ranges is only slightly more difficult than a single compaction step, it is much faster than computing a closure on the constraint graph. Using ranges for the y stretch rules has some drawbacks, though. Figure 4.15 shows an example of this method producing

[6] but worse than $O(v \cdot e)$. See Chapter 5 for discussion of all-pairs longest- or shortest- path algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more rigid stretch rules than the first method. More important, the ranges method is not always correct. Figures 4.16a and 4.16b shows an example of where it fails and produces an *unsafe* set



x
x_2
x_2
Unstretched cell boundaries
The ranges of x_1 and x_2 do not overlap
S
1
2
D

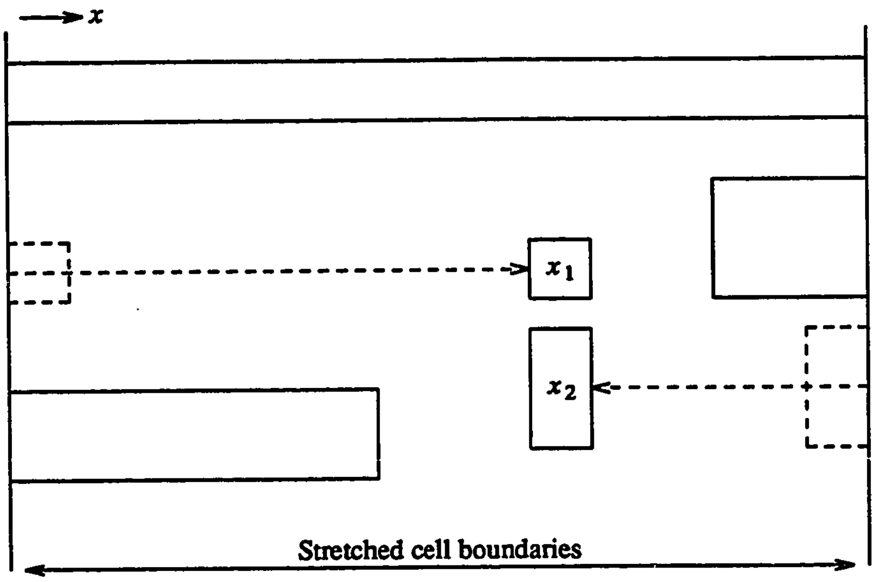
x constraint graph
Figure 4.16a Stretch rule generation: unstretched cell

of stretch rules.

To understand why the schemes work or fail refer to Figure 4.17. To simplify the discussion, assume the design-rule spacing between two squares of size two is zero. For the squares to be sufficiently close in x so as to require a y constraints, they must lie within region "R" in x_1-x_2 space. This region "R" is computed solely from the design rules and properties of the two components and not from their positions. It represents the set of positions of the squares such that they overlap in x[7]. When generating constraints for compaction, specific x values are

[7] Since R is an infinite band, the question *does a region intersect R?* is equivalent to *does the projection of a region intersect the projection of R onto a line orthogonal to R?* $x_1+x_2=0$ is such a line orthogonal to R.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

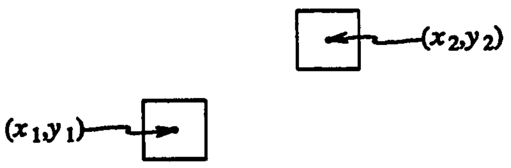


x-stretched cell: x_1 and x_2 can now collide in y!
Figure 4.16b Stretch rule generation: stretched cell

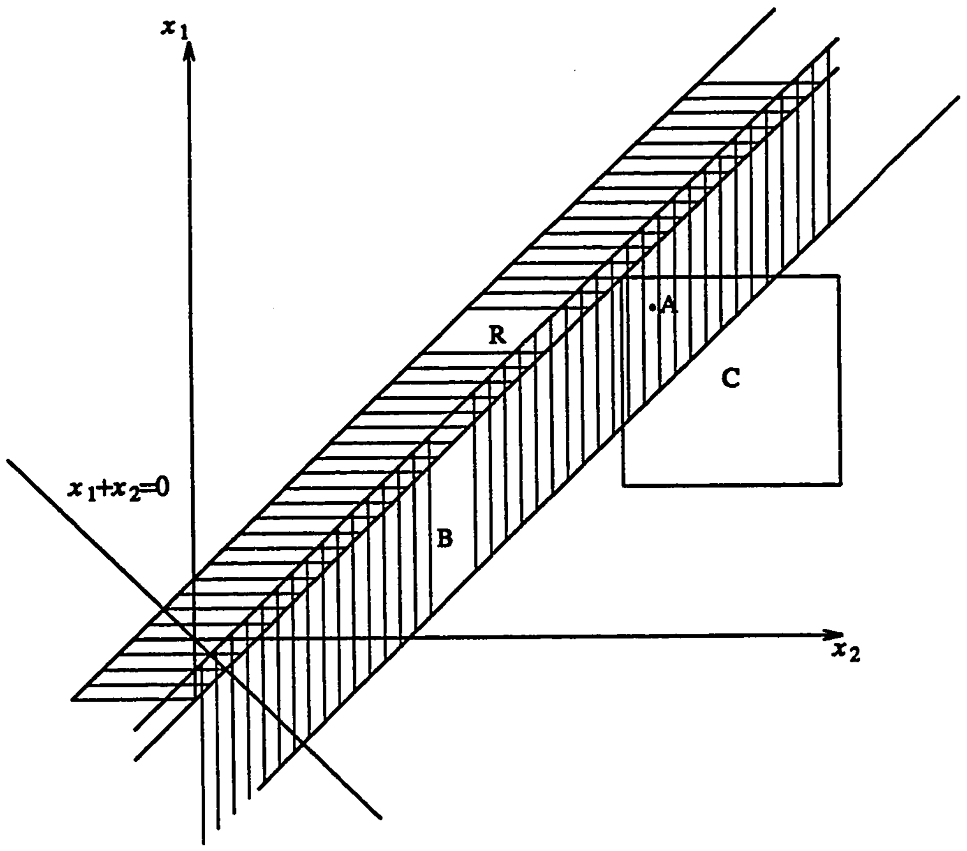
known, so the decision required is whether the point "A", representing the current values of x_1 and x_2, lies within region "R" (*i.e.* do the components overlap in x?). During stretch rule generation using the first (slow but correct) scheme, the actual values of x_1 and x_2 are unknown, but the region of their possible values is computed (region "B") and compared with region "R". In this example there is an overlap, indicating that a y constraint must be written. What happens in the second scheme? The ranges of possible values of x_1 and x_2 *in the compacted cell* are computed separately, producing a rectangular region C in x_1-x_2 space. Region C is an attempted upper bound on region B with respect to projection onto x_1+x_2=0. The scheme may fail if a y constraint is not generated because region C does does not intersect region R even though region B would have. The failure will only cause trouble when the cell is stretched. The stretching corresponds to a required enlargement of region C (the allowed range of x_1 and x_2 may increase when the cell stretches). Unfortunately, the constraints were generated with the region C computed on the basis of the unstretched cell. Although the error of this scheme is usually on the conservative side, sometimes, as shown, it is on the unsafe side.

This scheme can be easily repaired. If the ranges of motion in x appear to not overlap (thereby not producing a constraint in y), ensure that they cannot overlap by adding an additional

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



a) Can these rectangles overlap?



b) View in x_1–x_2 space

Figure 4.17 Deciding when to add a constraint for stretch rule generation

constraint in x that forces the x assumptions upon which the y constraints are based to be true. For every pair of objects, now, there exists a constraint in either x or y keeping them apart. The improvement just mentioned has not been implemented and so the data in Chapter 5 does not include the additional constraints. In practice, the failure of the algorithm implemented is very rare and was not discovered through its symptoms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstractions

In order to reduce the CPU time required to produce an IC layout, it is necessary to reduce the number of constraints in the description of a stretchable cell that needs to be manipulated at the next level up in the cell hierarchy. The only information that is required for a cell's instantiation is its external description — only its behavior or characteristics that can effect its surroundings or be determined from outside the cell. For Lava this external description includes a cell's bounding box, external connection points, and constraints on these connection points. This "black box" description is the cell's *abstraction*. The abstraction is a set of constraints that, if satisfied, guarantee that a design rule correct cell can be constructed.

The important information that must be in an abstraction comprises the constraints on how each external connection point may move in relationship to every other external connection point. These limits are the longest paths from each connection point to each other connection point. Thus the abstraction is a pair of constraint graphs (x and y) that have had all the vertices not associated with external connection points eliminated. All the arcs of the graph incident to the eliminated vertices have been summarized in arcs between the remaining vertices.

The abstractions for each cell in the chip are created by a bottom-up pass of the chip hierarchy. First all the leaf cells are compacted and their abstractions are created. Then for each level in the hierarchy, the constraints in the abstraction of a cell become the component constraints for that cell while its parent cells are being compacted and abstracted.

Stretching and Realization

At the point when all cells have been compacted and an abstraction produced for each, no actual positions have yet been decided, nor has geometry been produced. A top down pass over the chip hierarchy does the actual stretching of the cells, choosing final positions and generating the actual mask geometry. Beginning at the top cell, a pair of compaction steps produces final positions for all the geometry in that cell. Doing so choses the positions of the connection points

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for the immediate subcells. Each of these subcells, along with the positions of the connection points is placed on a queue of work to be done. Then the geometry for the top level is produced[8].

For each of the cells on the queue, a similar procedure to that performed on the top level cell is done. Before the two compaction steps are performed, the pin locations are constrained to match the connection points. These constraints fix the size of the bounding box and the positions of the connection points relative to the bounding box. Figure 4.18 shows the kind of constraints added during realization of a cell. This process continues until the queue is empty.

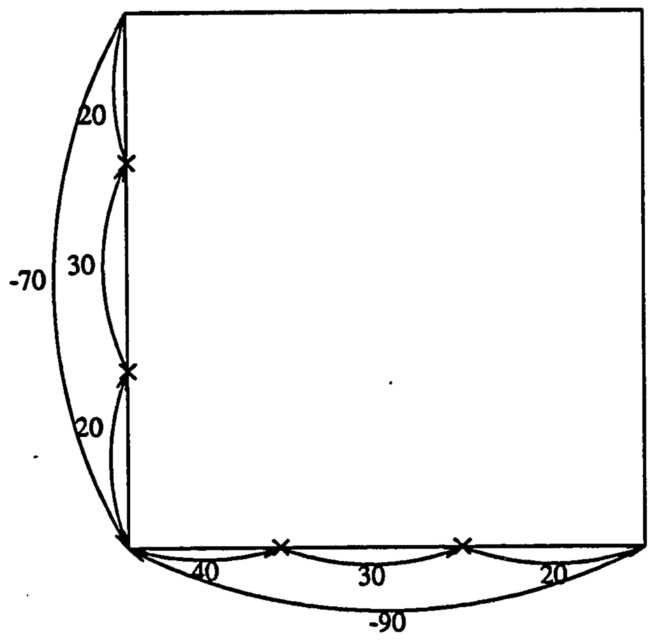


Figure 4.18 Added realization constraints

This basic method of using hierarchy makes symbolic layout feasible for use on entire IC designs. Both the CPU time and quality of the resulting layout need to be improved, however, to make these methods truly practical.

[8] An optimization that is used is that a cell can appear more than once on the queue if it is called more than once. However, multiple occurrences of the same cell with the the same set of relative positions of the connection points will all produce identical resulting geometry, so much time is saved by only placing on the queue cells with a unique set of connection point positions.

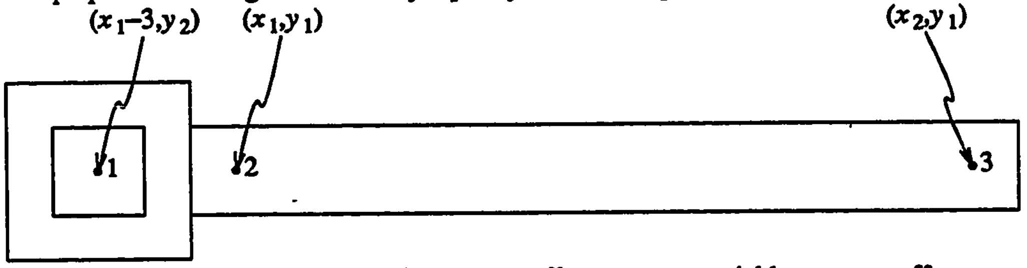
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1.3. Improvements and Extensions

Lava uses many techniques to reduce the CPU time complexity of the above algorithms or improve the quality of the resulting layout. These methods are described below.

Variable Merging

There are many cases where a pair of cyclic constraints is generated such that there is no slack or freedom of motion between the two components. This situation occurs when an equality constraint $x_i+C=x_j$ has been written as a pair of inequality constraints: $x_i+C \le x_j$ and $x_j-C \le x_i$. The optimization problem can be reduced in complexity by eliminating these extraneous vertices from the problem. During fixed constraint generation, the equality constraints are recorded[9] and then, before variable constraint generation, a mapping from component reference points to variables is prepared. See Figure 4.19. Any equality constraints produced during variable constraint generation are treated as before (a pair of inequality constraints is used) so that the mapping won't keep changing during compaction.



Reference point #	x variable	x offset	y variable	y offset
1	1	-3	2	0
2	1	0	1	0
3	2	0	1	0

Figure 4.19 Variable merging

[9] This task of maintaining equivalence classes (of variables) is a very well-studied problem known as the *union-find* problem and has some very efficient solutions [Aho 74].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Shadowing

Shadowing is a method of reducing the number of constraints in the constraint graph. The simplest method of generating the design-rule constraints produces a number of constraints that can be as poor as quadratic in the number of objects within a cell, since it may generate a constraint from every object to every other object. In a large cell, most of these constraints will be redundant. It is very important to eliminate a large portion of these redundant constraints, since the CPU and memory requirements will at best be quadratic if the number of constraints is quadratic. This section describes the shadowing method used by Lava to eliminate most of the redundant constraints.

Shadowing tries to improve two related problems: the amount of time constraint generation takes, and the size of the optimization problem generated. The simple constraint generation scheme takes $n^2/2$ time since it compares every component against every other component. It generates an optimization problem of between $O(n^2)$ and $O(n^{1.5})$ constraints, depending on the aspect ratio of the cell and the size of the objects as compared to the cell size[10].

How sparse of a constraint system is possible? By making some simplifying assumptions, we can derive an approximation. Assume that there are two non-interacting layers, which have only design-rule spacings between objects on similar layers: metal and polysilicon/diffusion. There are no design rules between objects on different layers. Further, assume that even when several objects on the same layer are electrically connected, a constraint is generated that prohibits them from passing through each other and interchanging positions. Under these assumptions, the subgraphs representing the necessary metal design-rule-spacing constraints and the necessary polysilicon/diffusion design-rule-spacing constraints will each be planar graphs. A planar simple graph with more than two vertices can never have more than $3v-6$ edges. The union of the two planar graphs, although not planar, will have no more than $6v-12$ edges[11]. Since the number of

10 If the cell is square and the objects small compared to the size of the cell, the exponent would be $O(n^{1.5})$ If the cell is a constant height, width n, x constraints being generated, and object are the full cell height, then $O(n^2)$ constraints would be generated .

11 Assuming that the graphs are connected (a good assumption) also eliminate a common spanning tree, reducing the bound to $5v-12$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

connection and component constraints is also linear in the number of components, the total number of edges in the constraint graph is linear.

How do the assumptions compare with reality? The second assumption could be made true with some loss of flexibility. In Lava, electrically equivalent objects can pass through each other, and a construct such as in Figure 4.20 requires n^2 design rule spacing constraints. Fortunately, such constructs with $n>2$ are rare enough to be insignificant.

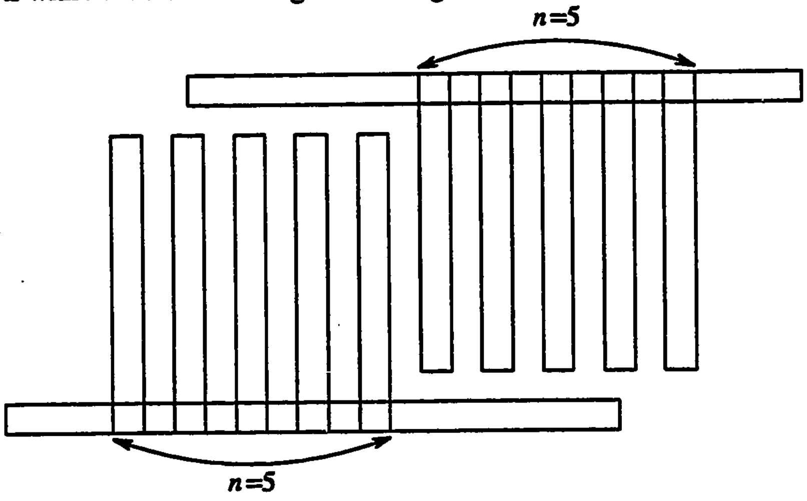


Figure 4.20 Construct requiring non-linear number of constraints

Therefore, in order to reduce the problem complexity, not only is it necessary to avoid generating constraints for all possible pairs of potentially conflicting components, it is also necessary to avoid examining all of these pairs. Can the sparse constraint graph be easily achieved? The following shadowing algorithm algorithm used by Lava does this.

Let us assume that constraints in x are being generated.

Step 1: Sort all components by their x coordinates.

Step 2: Initialize the set F to contain the left edge of the cell. Set F will contain components and fragments of components along with information (such as net number) that will allow determination of electrical connectivity. This set the *frontier*.

Step 3: For each component C in order of increasing x coordinate:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 3.1: Compare component C against each component or fragment in the frontier and write a design-rule-spacing constraint if applicable. (There is nothing novel about the constraints generated in this step – only that fewer component pairs are considered).

Step 3.2: Update the frontier. Every component or fragment in F that is "shadowed" by C is either clipped or eliminated from F. Component C is then added to the frontier set[12].

Under certain conditions, the shadow of C will be smaller than C. For example, when a polysilicon object shadows a diffusion object, the shadow must be reduced by $S_{DD}-S_{PD}$ because the polysilicon object now in the frontier cannot completely protect a diffusion object as yet unencountered. See Figure 4.21.

In order to estimate the amount of time that this algorithm takes, assume that the cell is square (a constant aspect ratio would be sufficient). The maximal size of the frontier would be $O(\sqrt{n})$, where n is the number of components. $n=O(x \cdot y)$). If a simple list is used as the data structure for F (there are better structures), then processing each component takes $O(\sqrt{n})$ time. Since there are n components, the algorithm takes $O(n^{3/2})$. There exist faster, but more complicated, algorithms which could have been used for computing this same set of constraints, but this algorithm was adequate for constraint generation.

While the above algorithm significantly reduces time and constraints during compaction, it is less useful for generating constraints for cell stretch rules. To use shadowing for the second direction of stretch rule generation, the algorithm must be modified. Instead of adding components to the frontier, a "smeared" component must be added. This smeared component is a union of the component in all its possible y positions. The shadow cast by a component is a lot smaller. The shadow is now the intersection of the component in all its feasible y positions. Therefore, if a component can move in y more than its y size, it casts no shadow and causes

12 Step 3.2 is a bit of a fib if electrically connected objects are allowed to pass through each other, as is the case in Lava. A piece of a component may only be removed from the frontier after it has been shadowed by two objects on distinct nets.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

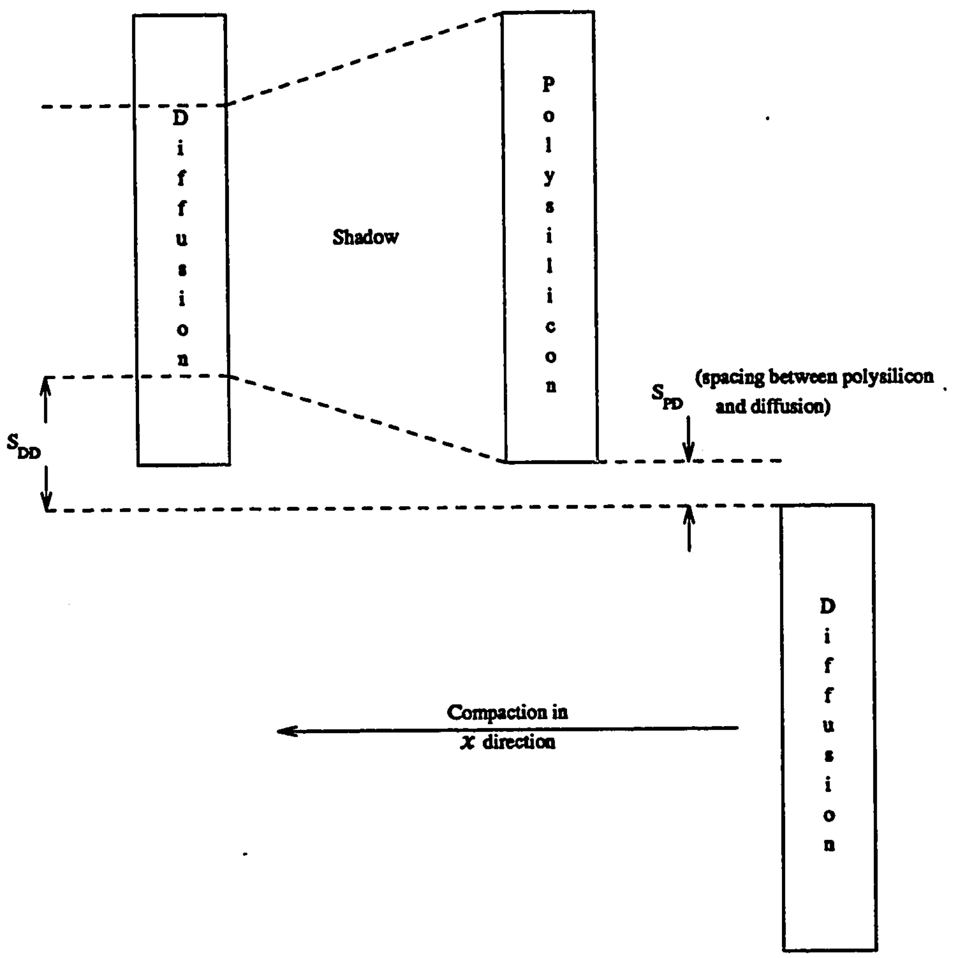


Figure 4.21 Conical Shadows

nothing to be eliminated from the frontier. As can be seen, large objects are being added to the frontier and little is being removed, resulting in a steadily increasing frontier size and much reduced performance.

The shadowing algorithm can be salvaged by realizing that the components that cast too small of a shadow by virtue of their possible motion are frequently adjoining similar components that move in concert. These components taken together will often cast a large enough shadow to be of use. See Figure 4.22. These improvements have not been implemented and are therefore not seen in the analysis of the next chapter.

Using a shadowing algorithm reduces both the size of the constraint system and the asymptotic time complexity of generating the constraints for all but the last step of stretch-rule

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

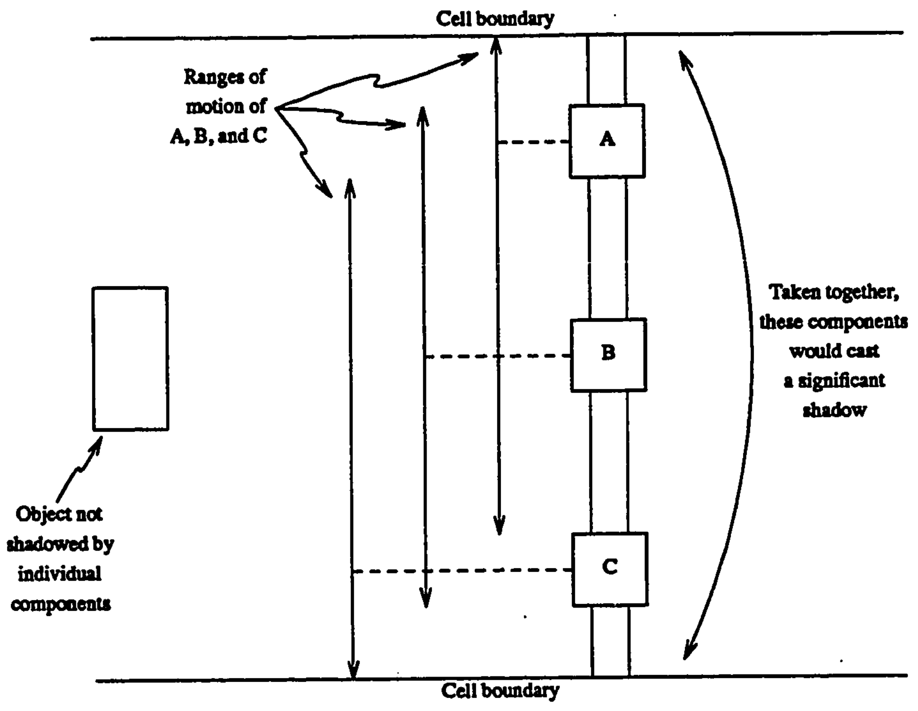

Cell boundary
Ranges of motion of A, B, and C
A
B
C
Taken together, these components would cast a significant shadow
Object not shadowed by individual components
Cell boundary

Figure 4.22 Second step stretch rule shadows

generation. Further improvements are also possible in the constraint-generation time for all steps and in the constraint system sparsity for the last step.

Wire Length Minimizations

After compaction in a given direction is completed, there are often components of a cell which can be moved without enlarging the cell. In many cases, it is very desirable to apply a secondary objective function to move these components to more preferable relative locations. Due to the iterative nature of compaction, such secondary objective functions have been found to make significant improvements in the final size of a cell. This section discusses these secondary optimizations, explains why they give cell size improvements, and explains why I chose to optimize weighted wire length.

Why do anything with the additional freedom? During compaction, sometimes components needlessly get in the way of other components and prevent the cell from compacting to an optimal size. In Figure 4.23, if object A were to move to the left, B moves down and reduces the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

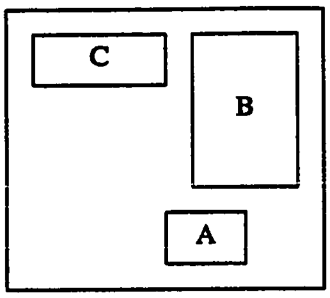


Figure 4.23 Larger than necessary cell

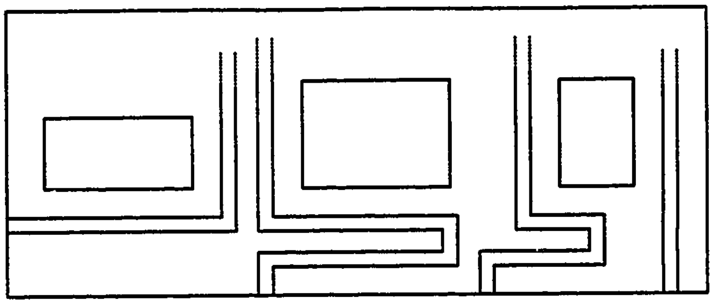
size of the cell. A much more intelligent algorithm (or a human) might recognize such a situation and do the "obvious" thing.

A number of simple heuristics were tried for using this extra freedom. Sometimes, the correct sequence of compactions in which the "loose" components are pushed all in one direction and then in another may move components out of each other's way and reduce the cell size. In the above example (Figure 4.23), compacting to the left before compacting down will reduce the cell size. It is not uncommon to derive some benefit from compacting the cell in each of the four directions (alternating x and y). This "shake, rattle, and roll" approach to cell compaction is very dependent on luck, as the operations being performed only have a chance relationship to the operations that need to be performed. Other heuristics, such as centering components in the middle of their range of motion, have similarly unpredictable success.

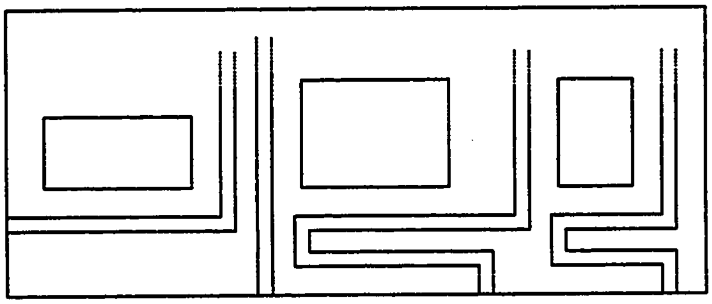
When there is more flexibility than necessary (additional jogs in the wires, for example), the problem only gets worse. In Figure 4.24, the wires with the jogs get in the way, and are too long under either a leftmost or a rightmost compaction. In Figure 4.25, the cell could become shorter if A would move to the right of B. None of the above compaction heuristics, without wire length minimization, will move A and B differently. Thus, the wires are needlessly long and the cell cannot be compacted further.

It should be clear that minimizing some function of wire length would be desirable from both a geometric (compaction) and an electrical standpoint. The question is *what metric should be optimized?* Several popular norms are the L^1 ($\sum_i |z_i|$), the L^2 ($\sqrt{\sum_i z_i^2}$), and the L^∞ ($\max_i z_i$).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



a) Right-most compaction



b) Left-most compaction

Figure 4.24 Extra Jogs

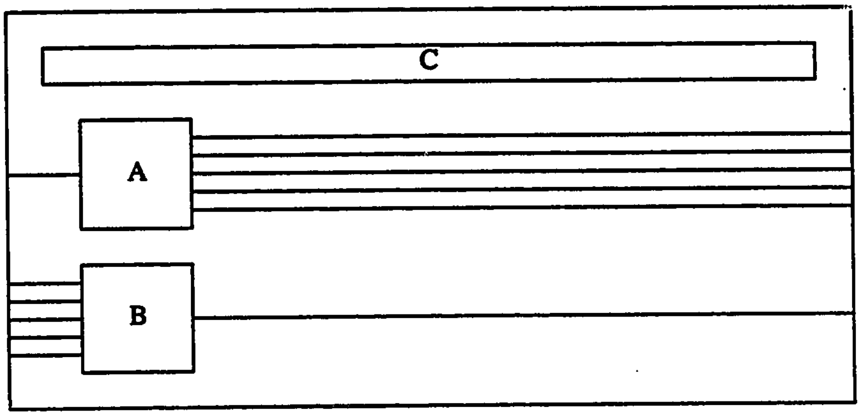


Figure 4.25 A compaction problem requiring wire-length minimization

All of these work equally well when applied to the example in Figure 4.24 because in this example there is no tradeoff between wires shortened and wires lengthened. Figure 4.26 shows each of the three norms minimized for the example of Figure 4.25.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

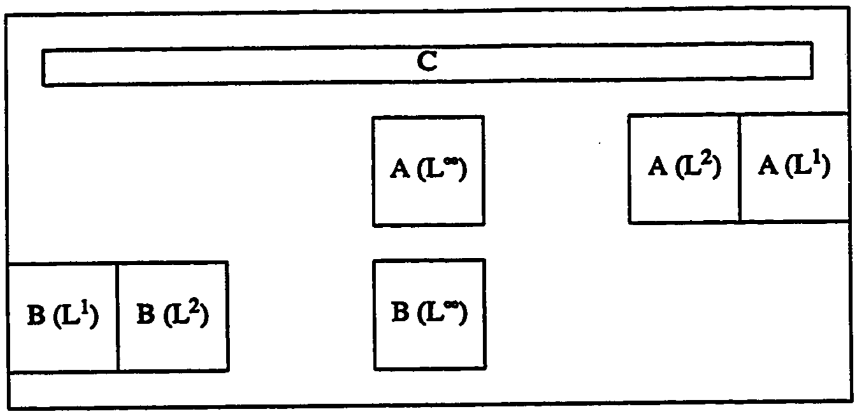


Figure 4.26 Results of three different norms applied to figure 4.25

The L^∞ norm, as can be seen, gives no consideration to the number of wires connected to an object. While this fact is not obviously poor from a geometric standpoint (despite Figure 4.26), it is very poor from an electrical standpoint. However, where the L^∞ norm really fails is when there exists a long wire that cannot be shortened due to constraints. In this case the objective function cannot be improved further by shortening any wire that is already shorter than the longest wire. See Figure 4.27 for an example of this problem.

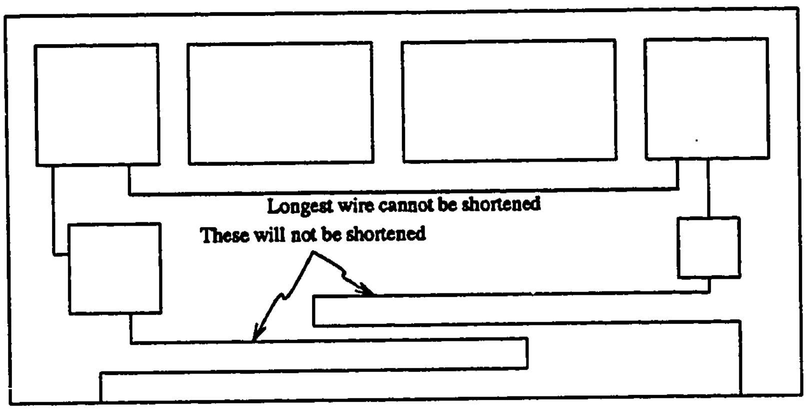


Figure 4.27 Shortcomings of L^∞

When using the L^2 norm, the wires act exactly like springs, pulling objects to each other. When these springs or wires oppose, a compromise on the position of the object is achieved depending on the number of wires on either side of the object. If the number on each side is identical, the object is placed exactly in the middle. From a geometric standpoint, right in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

middle is the place where the object is most likely to get in the way of further compaction operations. All the way to either side is usually better. From an electrical viewpoint, the L^2 norm will usually not minimize capacitance or delay, the first order effects of which are usually linear in position or dimensions. (The compactor, because it does not know otherwise, must assume all wires carry equally critical signals). An additional fault with the L^2 norm is that intuitively, two wires in series (Figure 4.28) should behave as one wire. This behavior does not occur with the L^2 norm.

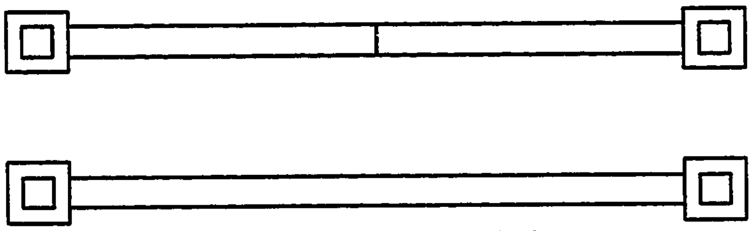


Figure 4.28 Concatenated wires

In choosing the position of an object, the L^1 norm behaves as if it were voting: the object moves closest to the side with the most wires. Objects rarely sit in the middle, impeding the compaction. Except for wires reversing direction, the L^1 norm is linear and therefore more closely matches the electrical properties that the designer is trying to minimize. Finally, only the L^1 norm will cause two wires in series behave identically to a single wire.

Because different wires have different degrees of criticality, weighting the lengths of wires differently might improve the layout. Signal wires could be given a greater weight than power and ground, diffusion wires (which have a high capacitance) could be weighted higher than metal, and the designer could, based on his knowledge, provide additional weighting information.

The entire optimization problem now becomes:

$$\textit{minimize}: \sum_k W_k \left| x_{i_k} - x_{j_k} + K_k \right|$$

$$\textit{subject to}: \left\{ x_i + C_{ij} \le x_j \right\}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

User Provided "Soft" Constraints

Lava allows the designer to specify his own constraints in addition to those generated by Lava. Because the user tends to be error-prone, robustness and error-diagnosis are enhanced by making these constraints *soft*. If an optimization problem has no feasible solution, the soft constraints are discarded as necessary in order to obtain a solution to the *hard* (Lava generated) constraints. This solution allows the designer to see what is happening to the cell and why his soft constraint could not be satisfied.

4.1.4. Summary and a Unified Perspective: An Optimization Function Hierarchy

Many of the constraints and objectives used to produce a cell layout are each written separately and solved sequentially. In this section I view these problems from a uniform perspective.

When solving the optimization problems during a compaction step, there are four objectives that are being optimized. In order of priority they are:

1) Satisfy all the hard (fixed and design-rule-spacing) constraints. These constraints may be rewritten as an objective as follows:

$$\begin{array}{ll} \textit{minimize} & \sum_k S_{h_k} \\ \textit{subject to} & x_i + C_{ij} \le x_j + S_{h_k}, \quad S_{h_k} \ge 0 \end{array}$$

If there exists a feasible solution to the hard constraints, the sum should vanish when optimized.

2) Satisfy all the soft constraints by minimizing $\sum_k S_{s_k}$ in a manner analogous to the hard constraints.

3) Compact the cell. Assuming $x_0=0$ (representing the left of bottom edge of the cell), the objective is to minimize x_1 (representing the right or top edge of the cell).

4) Minimize wire lengths:

$$\textit{minimize:} \ \sum_k W_k \left| x_{i_k} - x_{j_k} + K_k \right|$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Section 4.1, I described how to implement sticks compaction, and then showed how to extend it to be useful for entire chips by making use of hierarchy. Finally, I discussed some improvements which affect both the CPU time and quality of output and should allow Lava to be a practical tool.

4.2. Abut Cell Implementation: Tiling Algorithms

Sticks description is not necessarily optimal for describing all levels of an IC design. Sticks cells require pseudo-coordinates on all wires, and pseudo-coordinates and pseudo-sizes for all the subcells in the cell. These requirements result in aggregates (arrays and bundles) being sufficiently clumsy to use that little benefit is gained from them. Accordingly, this section presents the implementation of Abut cells.

The Abut cell type takes a different approach from Stix cells. The cell is assumed to be completely tiled by its subcells, and the relative positions of the subcells, when specified, are given by relations of the form *cell A is to the left of cell B.* No pseudo-coordinates or pseudo-sizes are ever needed. This section discusses the algorithms for implementation of this cell type. First, I discuss how a tiling is computed from the relations and abutting information, and then I discuss how the connection and abutting information is used to stretch the cells.

Tiling

The Abut-cell specification contains of the following information:

1) A list of subcells (along with their abstractions).

2) A list of interconnections between these subcells or between the subcells and the connection points on the boundary.

3) A list of relations between subcells of the form: cell A is leftof/rightof/above/below cell B. The meaning of leftof is that a vertical line may be drawn such that cell A is to the left of it and cell B is to the right of it. The other relations are analogous.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4) A list of relations of the form: cell A abuts leftof/rightof/above/below cell B. These relations imply the preceding relations (item 3) and additionally mean that if the cells are connected, the cells should be adjacent and the connections should be made by stretching and abutting the cells. Lava demotes this relation if the cells have no connections so that the designer can overspecify abut relations knowing that the nonsensical ones will be eliminated.

Connections without abut relations are set aside for a future wire router.

The information described above may or may not unambiguously specify a tiling of the cell. For example, the interconnection in Figure 4.29a, specified as

A abuts leftof B
C abuts leftof D
A abuts above C
B abuts above D

could cause a tiling as in either Figure 4.29b or Figure 4.29c. The tiling is computed by

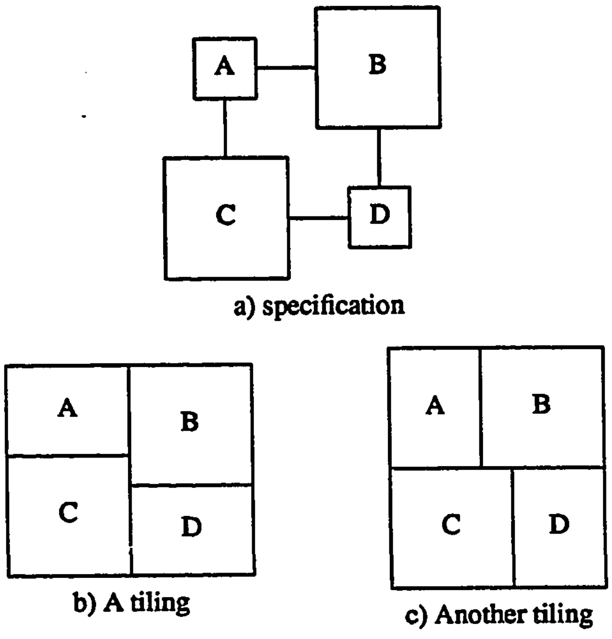


Figure 4.29 Ambiguity of Abut specifications

constructing and maintaining a system of linear constraints among the subcells, beginning with the user's relations. Inequalities are added to prevent cells from overlapping until a tiling is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

achieved such that no cell can overlap any other cell. At each step, an interrogation of the system of constraints determines which constraints can be added.

Let (x_0,y_0) and (x_1,y_1) represent the lower left and upper right corners of the cell being composed, and (x_{l_i},y_{l_i}) and (x_{u_i},y_{u_i}) represent the lower left and upper right corners on subcell i. The constraints included in the constraint system from input information are:

$$x_{l_i}+Xsize_i \le x_{u_i}, \textit{ and}$$
$$y_{l_i}+Ysize_i \le y_{u_i},$$

where $Xsize_i$ and $Ysize_i$ represent the compacted size of subcell i. On the assumption that the subcells are completely stretchable, realistic sizes are not strictly necessary, but including them is a heuristic that may produce a better tiling when there are several alternatives, resulting in less stretching. The constraints:

$$x_0 \le x_{l_i}, \quad x_{u_i} \le x_1,$$
$$y_0 \le y_{l_i}, \quad y_{u_i} \le y_1$$

constrain all subcells to remain within the boundaries of the cell being composed. For relations "cell i leftof cell j", the constraint

$$x_{u_i} \le x_{l_j}$$

is generated[13]. Analogous constraints are generated for the other relations. For the abut relation "cell i abuts leftof cell j", the constraints

$$x_{u_i}=x_{l_j},$$
$$y_{l_i}+C \le y_{u_j}, \text{ and}$$
$$y_{l_j}+C \le y_{u_i}$$

are generated. The first simply constraints the cells to be adjacent (in x). The last two insure that the opposing faces have a minimum overlap of C. Lava uses the minimum width of a connection for C, although C might better be a function of the number of interconnections between the two subcells. See Figure 4.30.

[13] For the purposes of illustration, the design rule spacing between subcells is assumed to be zero. This assumption could be true if the required space were included within the subcells. In any event, it simply reduces the number of nonzero constants in the constraints.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

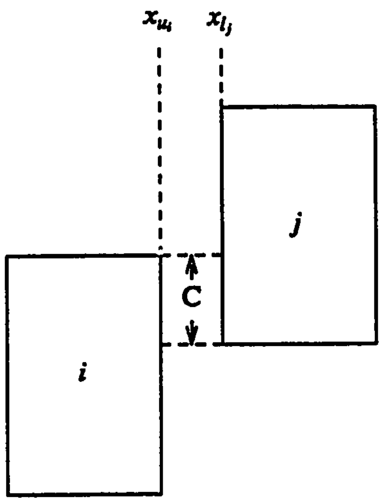


Figure 4.30 Abut constraints during tiling

For each pair of subcells we must ensure that they cannot overlap. To overlap, the subcells must be able to overlap in both the x and y coordinates. Therefore, to overlap all of the following statements must be feasible:

$$x_{u_i} > x_{l_j}$$
$$x_{u_j} > x_{l_i}$$
$$y_{u_i} > y_{l_j}$$
$$y_{u_j} > y_{l_i}$$

These conditions can be tested as follows.

The type of question that will be asked of the constraint systems is: can $x_a < x_b$ or $y_a < y_b$?. This question is used to determine whether two cells may overlap, and if so, what constraint may be added to prevent it. To answer the question efficiently, a closure of the constraint system is computed[14]. To determine if $x_a + k_1 < x_b$ is feasible (possible for some x_a and x_b), find in the system the constraint $x_b + k_2 \le x_a$. If the second constraint does not exist or if $k_1 + k_2 < 0$, then the former constraint is feasible.

[14] In the closure of a constraint system, if $x_1 + a \le x_2$ and $x_2 + b \le x_3$ are in the system, so is $x_1 + c \le x_3$, where $c \ge a + b$ and c is the length of the longest path from vertex 1 to vertex 3. Computing the closure is the same as computing all pairs of longest paths. If the closure cannot be computed (some of the arcs have infinite length), then the original problem was inconsistent. This might result from a specification such as "A leftof B leftof A".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the subcells can overlap, one of the following constraints must be added to the system:

$$x_{u_i} \le x_{l_j}$$
$$x_{u_j} \le x_{l_i}$$
$$y_{u_i} \le y_{l_j}$$
$$y_{u_j} \le y_{l_i}$$

Note that these constraints correspond to the constraints generated by the relations. Not every one of the above four constraints can be added while retaining a consistent set of inequalities. They must be tested and one of the feasible constraints chosen and added to the constraint system. An algorithm called *constraint propagation* (to be described in the next chapter) is used to propagate the effects of the added constraint throughout the closure.

The tiling algorithm continues testing each pair of cells and, if necessary, adding constraints until it either all cells have been tested or it encounters a pair of cells for which none of the four possible relationships can be added to the system (meaning the cells are constrained to overlap). This problem is due to a relationship having been assumed which was a poor choice. This failure tends to only occur for quite ambiguous descriptions. When the sequence of relationships assumed by Lava is displayed to the designer, the solution becomes obvious. Some assumed relations were inconsistent with the designer's intent, so he can fix the problem by adding more relations to better express his intent.

Abut Cell Stretching

In the previous section a set of constraints was computed to represent a set of positions of the subcells. The constraints guarantee that no overlap of the subcells may take place and that subcell faces that must abut are adjacent. As yet, neither the subcell's abstractions nor the interconnections have been taken into account. This section describes how these additional requirements are included.

The system of inequalities we have been working with so far only has variables representing the edges of the cell and its subcells. To this system, variables representing all of the connection points must be added.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The constraints of the abstractions representing each of the subcells are now merged into the constraint system. The system of inequalities will remain consistent in all cases if the subcells are stretchable, and in most cases otherwise. Figure 4.31 shows an example of an incorrect layout description, which can only be detected at this point. C is as large as A, A is not

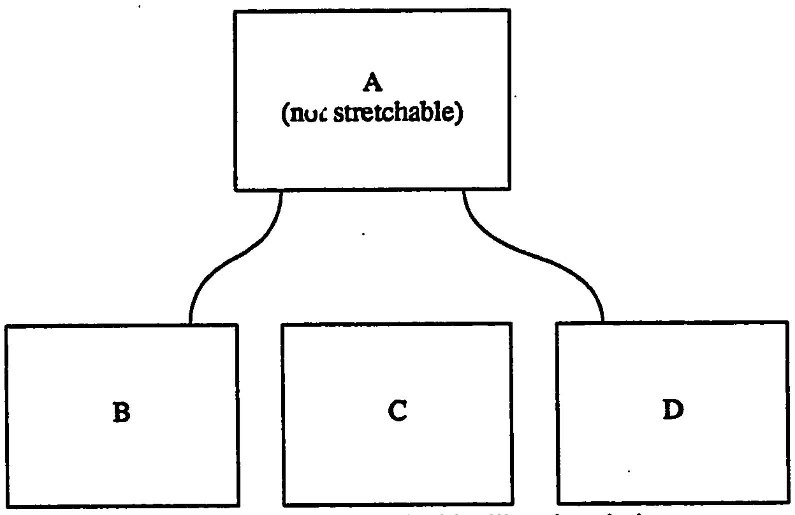


Figure 4.31 An unresolvable tiling description

stretchable, and the specifications included "A abuts above B" and "A abuts above D".

Next, the constraints representing the connections are added to the system. These constraints are the same type as the connection constraints used in the Stix cell type. Because the wire segments used to connect the subcells together are short (exactly the cell-to-cell design-rule spacing in length), and because the wire segment may be no wider than the connection point on the subcell, which has already taken design-rule spacing into account, no design-rule spacing constraints need be determined between the wire segments or between the wire segments and the subcells[15]. The subcell-to-subcell separations have already been taken care of by the constraints generated during the tiling phase.

At this point the constraint system is solved for the longest path either from the left/bottom edge, or the the right/top edge just as during compaction. The resulting coordinates will work as

[15] Note that at this point I have revoked the assumption about a zero cell to cell design-rule separation and am now using the maximum design-rule separation between layers (which happens to be the metal-to-metal rule for Mead and Conway design rules) as Lava does.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pseudo-coordinates that can be given to the compactor. Alternatively, the x and y constraint systems can be used directly to build stretch rules and an abstraction[16]. Constructing the stretch rules is extremely easy: there were no variable constraints at all so the constraint systems are the stretch rules. There does not exist an interdependency between x and y. Because of the lack of variable constraints in abut cells, computing the abstractions (and instantiating them) is a much less complex problem.

4.3. Summary

I began this chapter by describing a simple sticks compactor. I then extended it to permit the use of hierarchy in the IC description by creating abstraction of stretchable cells. This extension alone would allow symbolic layout to be used for the design of entire chips. However, further improvements such as variable merging, shadowing, and wire-length minimization improve the quality of the layout or reduce the time required to produce the layout in order to make Lava a practical IC layout tool. Finally, I describe the compilation of the Abut cell type. Abut cell types use a slightly ambiguous description of the cell, which is easier for the designer to produce and also can be faster to compile.

In this chapter I have described the essence of the algorithms that compile Lava cells into three optimization problems in such a manner that the optimization problems are relatively small and easy to solve. How small and easy to solve is the subject of the next chapter, which studies the optimization problems and the solution algorithms.

[16] Lava does not do this more efficient approach simply as a matter of programming expediency. It was easier to just reduce the abut cell type to a previously solved problem: sticks compaction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Solution of Optimization Problems

In the previous chapter I discussed compiling a Lava description into a set of optimization problems. Preference was given for small and sparse optimization problems. This chapter discusses solution methods for the optimization problems generated by the algorithms of the previous chapter. First, I will begin by analyzing and characterizing the problems. Such measurements tell us how well the constraint generation algorithms worked and help us choose appropriate solution methods. Next, I develop algorithms for the solution of optimization problems based on these analyses. Finally, I present and analyze CPU time performance measurements for these optimization algorithms and for Lava in its entirety.

5.1. Problem Characterization

The choice of algorithm is affected by the size of the typical problems, the sparsity of the constraint graph, and properties such as the structure of the graph and how it relates to decomposability. For example, if all problems are small, the asymptotic time complexity of the solution algorithm becomes less important than the actual time for small problems. The worst-case constraint graph might have v^2 edges[1], where v is the number of variables in the problem and also the number of vertices in the constraint graph, for which the fastest solution algorithms are typically worse than $O(v^3)$. Fortunately, this worst case does not occur. The constraint graphs are typically sparse with the number of edges proportional to $v^{1.3}$. The following subsections discuss the measured characteristics of optimization problems that were generated in the compilation of two different IC layouts. One layout is a tester controller chip ("TC") with 1633 transistors (including PLA) and the other is the data path section from a floating point processor chip ("FP") with 3499 Transistors.

[1] I am assuming only simple graphs are permitted. To reduce a graph to a simple graph, replace all parallel edges by a single edge whose arc length is the maximum of the arc lengths.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1.1. Problem Size

The first characteristic to examine when studying a problem is its size, which for our problems is the number of vertices (v) in the constraint graph. This number is closely related to the complexity of the IC layout. It varies linearly with the total number of components, wires, and terminals in the cell layout[2]. Also, the size of the solution to the optimization problem we seek is of size v.

How does v compare to cell area as a measure of problem size? Because the average size of the components and wires increases as the size of the cell increases, v will grow slower than the cell area. Two extremes will form useful bounds on the growth of v. If we assume that the average rectangle area is constant, then v is proportional to the area of the cell. If, on the other hand, we assume that the average rectangle size (area) grows linearly in the linear size of the cell (as would happen if a constant fraction of the rectangles were wires crossing the cell), then one could expect v to be $O(\sqrt{area})$ for a constant aspect ratio. If we can separate the rectangles into two classes, those which due to their function (such as cross chip communication) would tend to increase in average size as the chip grows, and those whose average size is constant (logic circuits), and if a non-decreasing proportion (k) of rectangles are in the latter category, then we would expect v to be $k\,O(area) + (1-k)\,O(\sqrt{area}) = O(area)$. Since I am using v as a measure of problem size, the results of this section should appear more pessimistic than if area were used as a measure of problem size.

Subcells don't change the picture much. One make make a variety of assumptions about how the number and size of subcells grows with chip area, ranging from the number of subcells per cell being constant ($O(1)$), with size $O(area)$, to the number being $O(area)$ with constant size. The number of vertices that each subcell brings into the picture is roughly the number of connection points it has, which, because they must fit on the boundary, we assume is bounded by

[2] This linear dependence is because each item (components, wires, and terminals) has a small (typically one in each dimension), constant (independent of the number of items) number of variables. There is at most one variable for each dimension for each separately movable portion of a component. The actual number of variables is less because many variables are eliminated due to equality constraints arising from connections.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the square root of its area. Putting these assumptions together, the total number of vertices contributed by subcells ranges from $O(\sqrt{area})$ to $O(area)$, respectively.

Therefore, while v is not proportional to area, and initially grows slower than area, we would expect v to be asymptotically linear in area.

The majority of the constraint problems in the two layouts studied are small. The problems range in size from 3 vertices to 824 vertices with 50% under 32 and 40 vertices in "FP" and "TC", respectively. See figures 5.1a and 5.1b for the size distributions. The small problem sizes

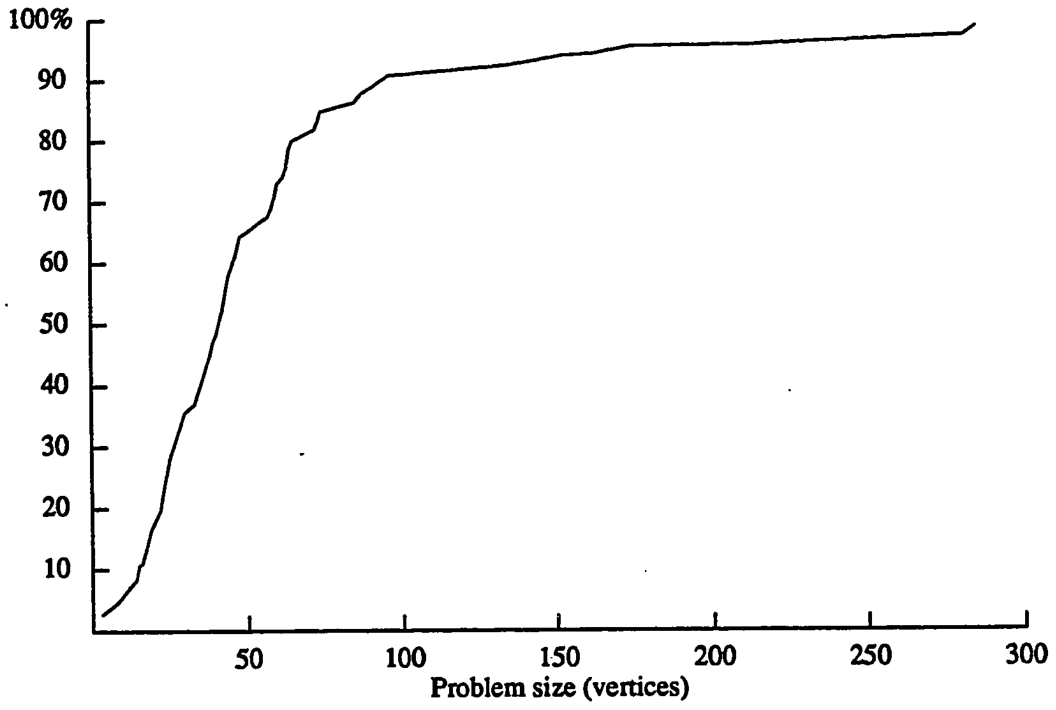


Figure 5.1a Cumulative distribution of problem sizes for TC

suggest that there exists great potential CPU time efficiency gains due to the designer's use of hierarchy in these chips and due to the design tool's exploitation of the hierarchy.

5.1.2. Sparseness

Density is another important characteristic of the constraint graph. As mentioned above, the worst problems may have v^2 edges in the graph, with a correspondingly poor solution time. Fortunately, the problems that occur in practice are rarely this dense. For sparse problems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

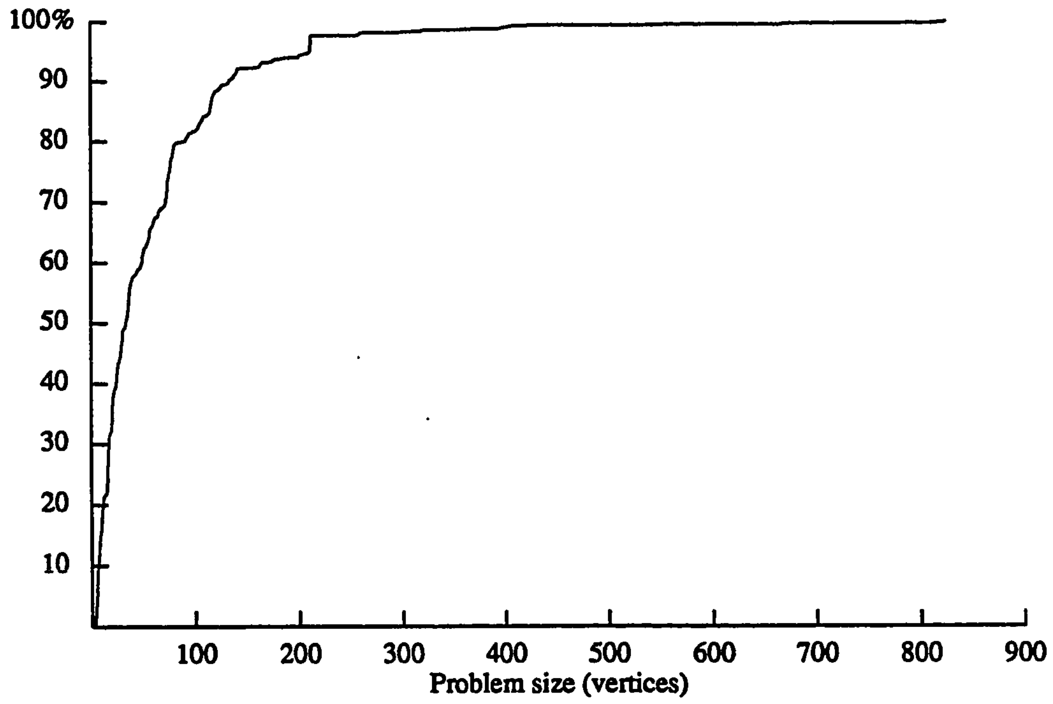


Figure 5.1b Cumulative distribution of problem sizes for FP

algorithms exist whose time complexity depends on, and is not much worse than linear in, the number of edges. These algorithms will be discussed in §5.2.

As can be seen from the scatter plots in figures 5.2a though 5.2d, the number of edges, e, is considerably less than v^2 and the density (as a percentage of a complete graph) is a decreasing function of v.

In order to understand how the number of edges depends on the number of vertices I tried fitting the data to several simple models using least squares estimation[3]. Fitting a second order

[3] Least squares estimation chooses parameters for a function f(x) so as to minimize the sum squared estimation error:

$$\sum (y_i - f(x_i))^2.$$

For fitting to a polynomial $x=v$ and $y=e$; for fitting to an exponential $x=\log v$ and $y=\log e$. A measure of goodness of fit is also computed, which is

$$1 - \frac{\sum (Error)^2}{\sum (y-\bar{y})^2}$$

A value of zero is uncorrelated noise, and one is a perfect fit. For fitting $y=C_0+C_1x$, the measure of fit is the same as the square of correlation coefficient.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

polynomial:[4]

$$e = c_1 v + c_2 v^2$$

arrives at the coefficients shown in the table below, suggesting that e is not quadratic in v.

	c_1	c_2	c_1/c_2	Fit
TC	4.33	.018	236	.66
FP	8.45	.0078	1076	.65

The ratio c_1/c_2 is the v for which the linear and quadratic terms are of equal magnitude. Note that in either case this number is around the upper end of the range of the data.

Fitting to

$$e = K_1 v^{k_2}$$

gives an estimate of how e depends upon v. For TC, e=1.34$v^{1.34}$ (fit .94), and for FP, e=1.26$v^{1.32}$ (fit .95). These curves are close to each other and appear to fit the data reasonably well.

The constraint problems have several different sources which one would expect, for theoretical reasons, to have different asymptotic behavior. The different sources are as follows (refer to Chapter 4 for details of the constraint generation algorithms).

(''I'')Initial correct layout generation. Indicated by an 'X' in the scatter plots, the constraints for these problems consist of component and connection constraints, plus one additional constraint per component. There are no design-rule-spacing constraints, so the number of edges should be linear in v.

(''A'')Abut cell type, indicated by a diamond in the scatter plots. These problems consist of connection and component constraints plus $m^2/2$ additional constraints, where m is the number of subcells. One would expect the number of subcells to be independent of cell size, whereas the number of connections and wires would increase with the size of the cell and subcells such that the $m^2/2$ term is not significant. The number of edges should be only slightly worse than linear.

[4] The constant term is omitted because the curve must go through (or at least close to) the origin: we know that $v-1 \leq e \leq v^2$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

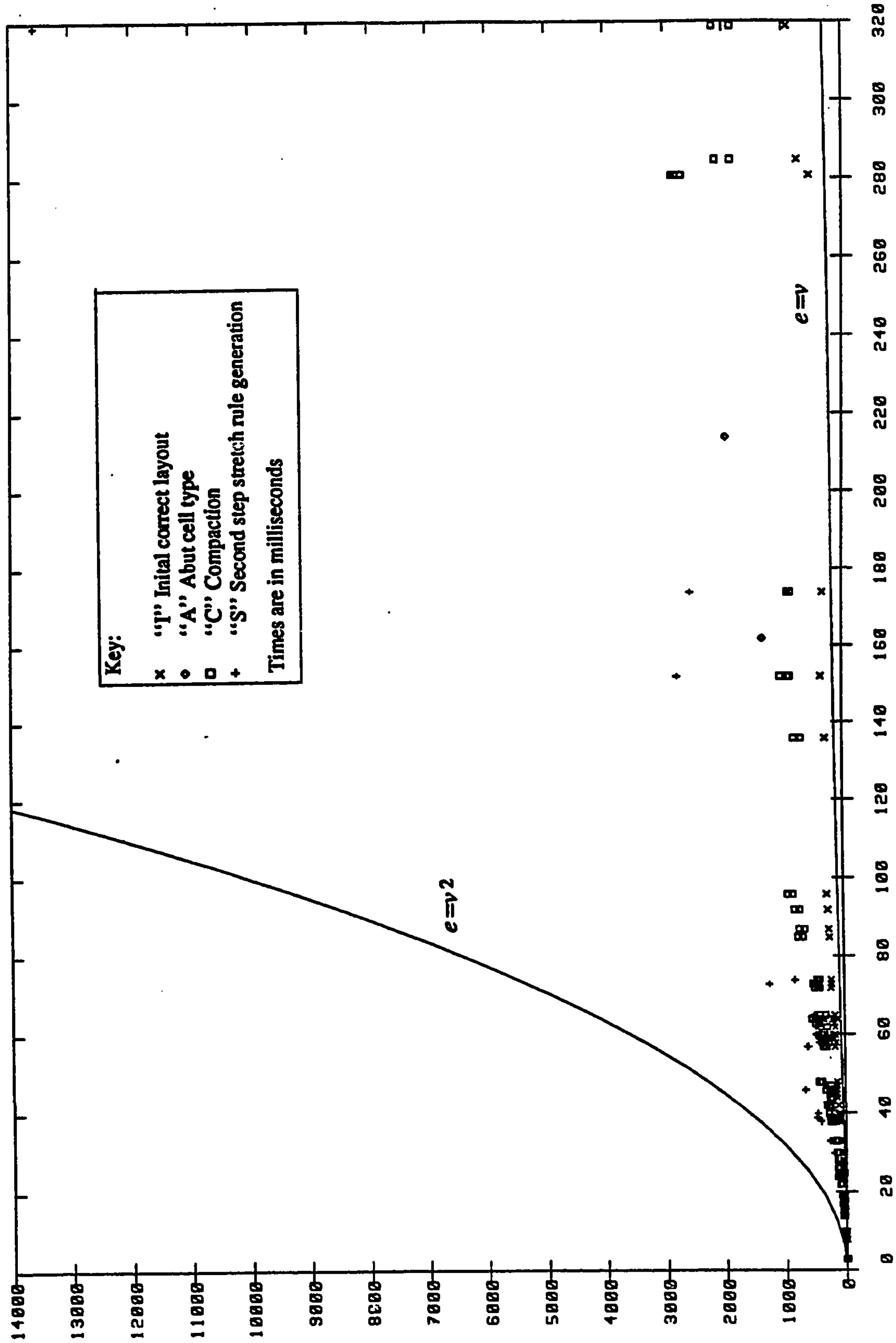


Figure 5.2a edges vs. vertices for "TC" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

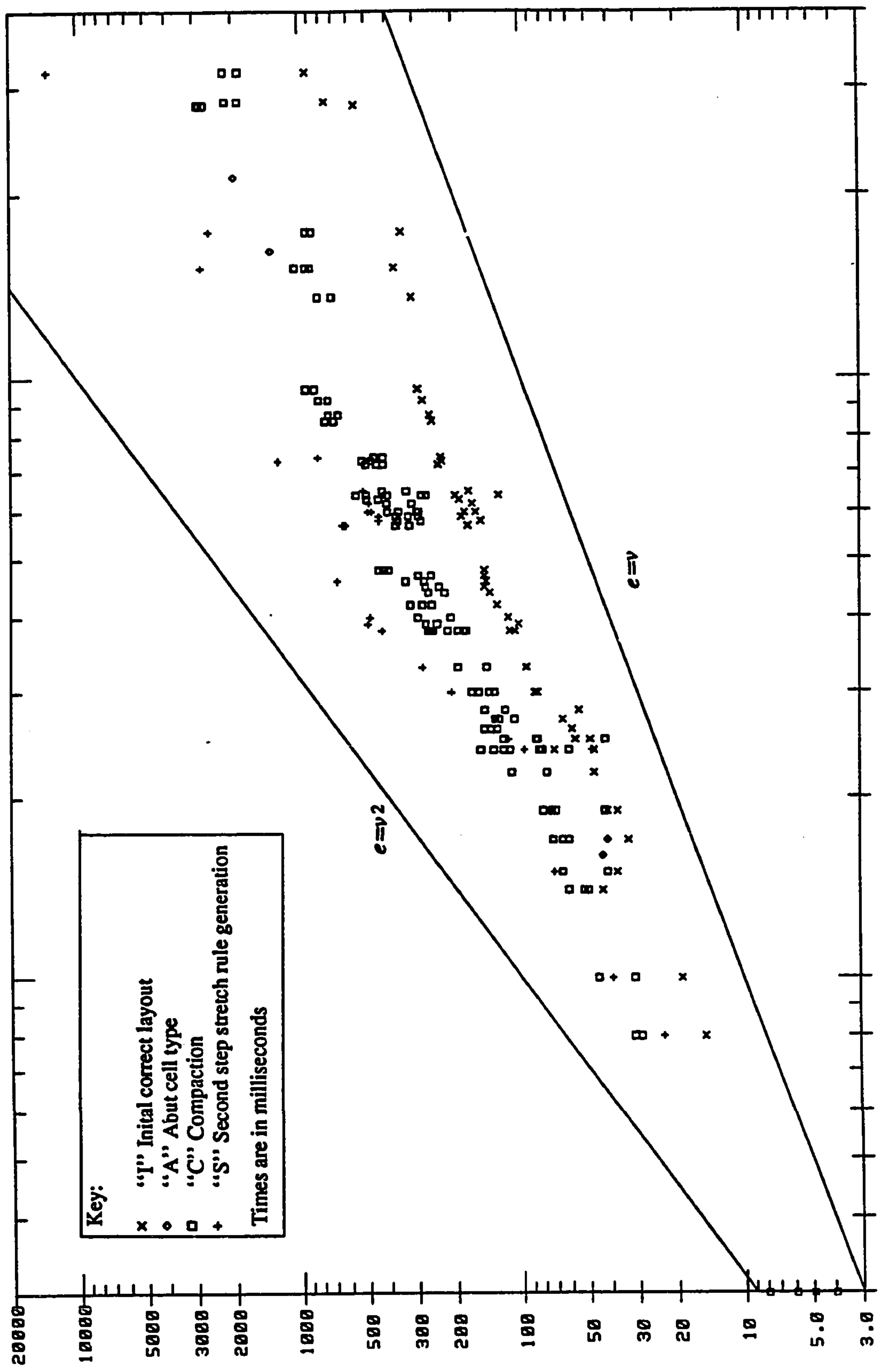


Figure 5.2b edges vs. vertices for "TC" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

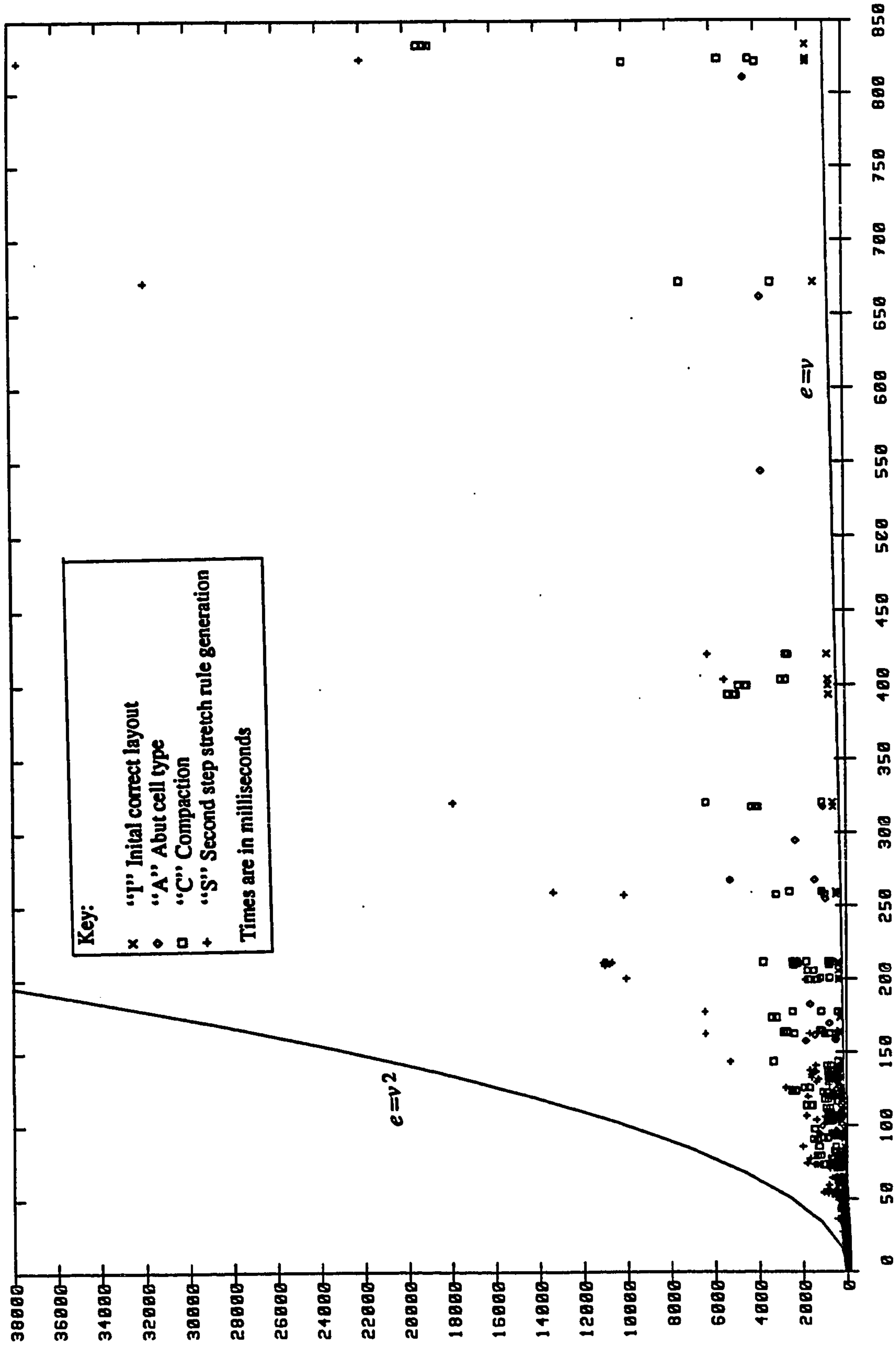


Figure 5.2c edges vs. vertices for "FP" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

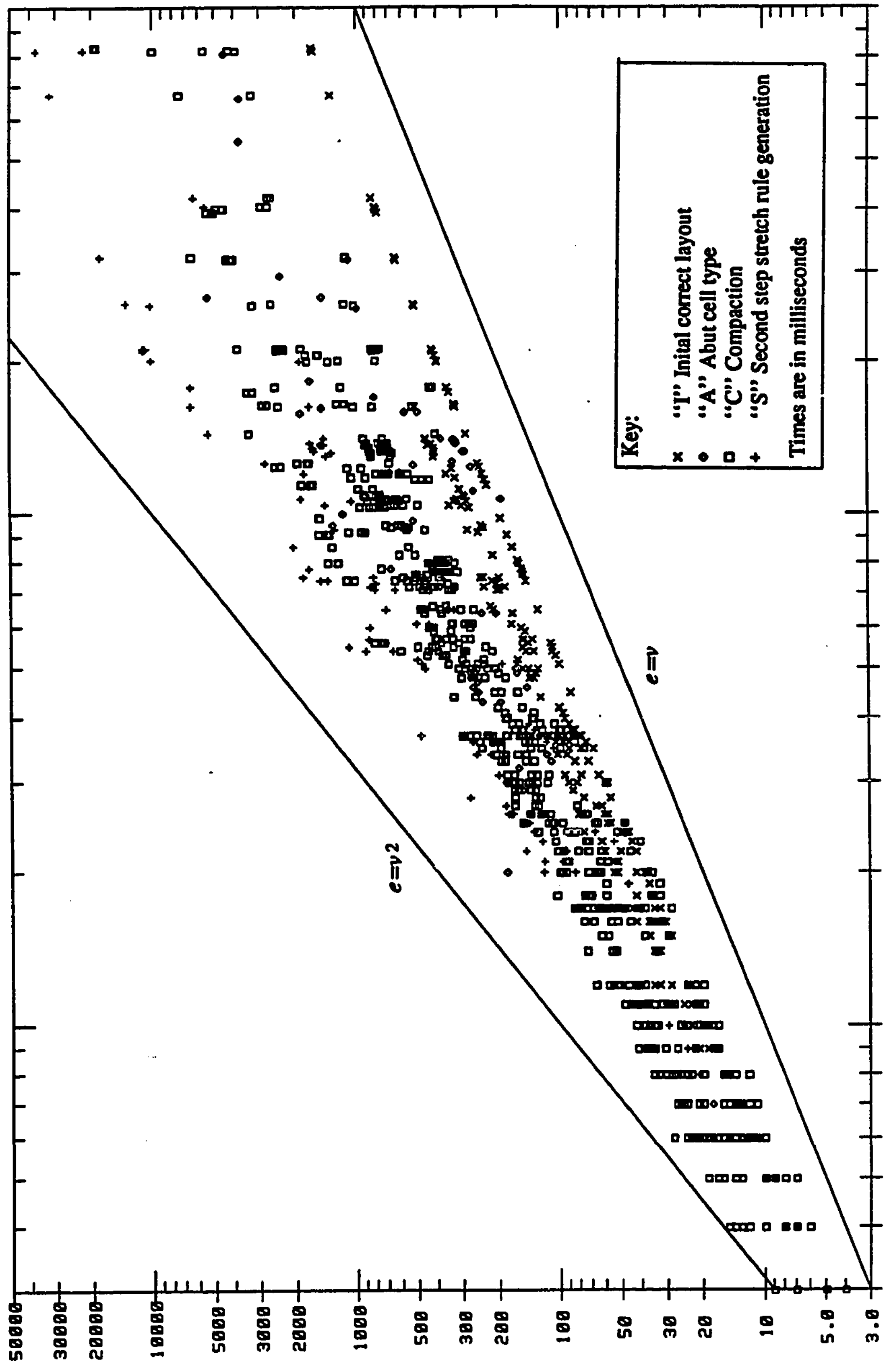


Figure 5.2d edges vs. vertices for "FP" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

("C")Compaction steps with shadowing, indicated by a box in the scatter plots. This category includes all other compaction problems except the last step of stretch-rule generation. These problems consist of connection and component constraints, as before, plus design rule spacing constraints. If an optimal job of constraint generation with shadowing is performed, the number of constraints generated will, in most cases, be linear in the number of vertices[5]. However, such an optimal job is difficult and probably not worth the cost in CPU time. A different algorithm is used which produces good, but suboptimal, results. As discussed in Chapter 4, there are certain constructs that will cause the number of constraints to be quadratic in the size of the construct. In actuality, these constructs are quite small. Therefore, the number of constraints generated typically should be somewhat worse than linear, but much better than quadratic.

("S")The last step of stretch rule generation indicated by a '+' in the scatter plots, cannot use the same shadowing algorithm (as discussed in Chapter 4). Instead, a simple quadratic (in CPU time) constraint generation algorithm is used. While the worst case number of edges could be quadratic in v, a number of factors (such as aspect ratio) can reduce the number of constraints such that empirically one would expect slightly better than quadratic behavior.

The results from this decomposition by problem source are as follows:

		$e=c_1v+c_2v^2$			$e=K_1v^{K_2}$		
		c_1	c_2	Fit	K_1	K_2	Fit
TC	I	2.75	-.00070	.976	1.76	1.10	.986
	A*	4.6	.022	.999	.61	1.5	.999
	C	5.57	.0050	.956	1.14	1.38	.976
	S	-0.41	.13	.992	.72	1.64	.975
	All	4.33	.018	.664	1.34	1.34	.944
FP	I	2.21	-.0003	.988	1.89	1.04	.992
	A	4.97	-.00014	.833	3.1	1.06	.897
	C	4.23	.0074	.788	1.15	1.32	.980
	S	37.3	.0014	.910	.42	1.80	.980
	All	8.45	.0078	.648	1.26	1.32	.948

* there were only 4 data points in this category.

The exponents of v (K_2) in the exponential fit coincide fairly well with expectations.

[5] See discussion of shadowing in Chapter 4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1.3. Problem Decomposition

Another important property of the constraint graph is how easily it may be decomposed into subproblems which may be separately solved. If the optimization problems have a structure such that they can be easily decomposed, the subproblems solved, and these solutions easily recombined, the speed of solution may be improved. In this section I consider the structure of the problems with respect to decomposition for solution. First I will discuss decomposition without wire length minimization (*spring arcs* in the graph). Then I will discuss the effect spring arcs have on decomposition.

If the graph were acyclic, there exists a solution method (CPM or PERT) for which the CPU time is $O(v+e)$. These solution methods, which use the acyclic nature of a problem graph, are a special case of solution methods that use decomposition into strongly connected components (SCC). In this case (acyclic graph), each of the SCCs are a single vertex. Unfortunately, Lava rarely generates an acyclic constraint graph. Decomposition into SCCs takes $O(v+e)$ time and composition of the subsolutions takes time proportional to the number of intercomponent edges. Therefore, the more (and smaller) SCCs there are, the better (and closer to $O(v+e)$) the solution becomes.

Many of the decomposition methods yield a partition of the graph into subcomponents and an order in which the subcomponents and the edges interconnecting them must be processed to correctly propagate longest-path information across the graph. For instance, if the graph is acyclic, it can be solved by propagating the longest path from the source vertex along the edges in a single pass in topological sort order (the components are single vertices and therefore require no processing). If the graph is cyclic, but decomposable into SCCs, the components and intercomponents edges are processed in topological sort order. The edges interior to a component may need to be traversed more than once, however.

How decomposable are the problems? All problems that are generated during the realization[6] phase consist of a single large SCC due to the constraints added to keep connection points on the cell boundary (and the boundary itself) from changing to something other than that used in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the composition of the parent cell. All the other problems (from the abstraction phase), however, tend to have many small SCCs. In FP, 35.6% of the problems are acyclic (maximum component size of one), 67.2% have a maximum component size less than or equal to three, 98.6% are ≤ 15, and the largest component consists of 34 vertices. Looking at this information another way: the average component size ranges from 1.00 to 7.71 and 98.1% have an average component size below three.

Abstraction SCCs		TC	FP
Max component size		156	34
Percent of problems with max. component size less than	1	14.0%	35.6%
	3	33.0	67.2
	15	92.6	98.6
Avg. component size		1.84	1.46

The abstraction problems are, therefore, quite easily decomposable.

What about the realization problems? A more powerful decomposition technique is needed for these. Expectedly, many of these problems with a single strong component are very similar in structure to problems with many components, but merely have a small number of back arcs added to the graph.

Consider the case where a single back arc is added from the vertex representing the right or upper edge (the sink of the graph) to the vertex representing the left or bottom edge (the source). This edge will cause the graph to consist of only one strong component. If a solution to this new problem exists, it is the same as the solution to the problem without the added edge. The additional edge only affects the existence of a solution. One would suspect then that there might exist more powerful decomposition techniques that would allow us to use the structure which we know is there.

[6] As discussed in Chapter 4, there are two major phases to the layout translation: first, a bottom up *abstraction* phase, where abstractions of the cells are created to be used at the next level up in the cell hierarchy; and a top down *realization* phase, where final positions are chosen for all the terminals and components and the geometry is generated. The constraint problems are very similar except for some additional constraints (during realization) representing the choice of terminal locations specified by the immediately superior cell.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A more powerful decomposition technique based on reducible[7] flow graphs does exist [Tarjan81a]. While the example above, which is not decomposable into SCCs, is *reducible*, and therefore could be decomposed by this technique, in general, most of Lava's constraint problems that are not decomposable by SCCs are not reducible either.

There exists another decomposition technique which is based on the dominators of the graph [Lengauer 79]. This method decomposes the graph into *dominator strong components* (DSC) [Tarjan81a]. DSCs have an analogous relationship to reducible graphs as SCCs have to acyclic graphs, that is if a graph is reducible, all the DSCs are single vertices. DSC decomposition decomposes the realization problems as successfully as SCC decomposition decomposes the abstraction problems.

All DSC's		TC	FP
Max component size		156	34
Percent of problems with max. component size less than	1	11.9%	28.3%
	3	31.8	66.0
	15	93.5	98.7
Avg. component size		1.83	1.49

DSC never decomposes an abstraction problem any further than SCC decomposition does. These fundamental components to which SCC and DSC will break down the problems to appear to be mostly caused by connection constraints as illustrated in figure 5.3a, resulting in graphs like that of figure 5.3b. These sliding connection constraints are the major cause of the acyclic nature of the constraint graphs in Lava.

The addition of spring arcs (L^1 minimization) make the decomposition problem more difficult. In nearly all of the problems that have spring arcs, 100% of these spring arcs go between vertices in different strong components (or DSCs where there is only one strong component). Because of the two way nature of the spring arcs (they both push and pull, unlike regular constraints which only push), there does not appear to be a simple decomposition technique

[7] A reducible graph is one that can be transformed into a single vertex by the following transformations: 1) Removing a loop: an edge from a vertex to that same vertex is removed. 2) Merging vertices: if all edges entering vertex v leave vertex w, merge vertices v and w. Tarjan writes "Intuitively, a flow graph is reducible if every cycle has a single entry from the start vertex" [Tarjan81a].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

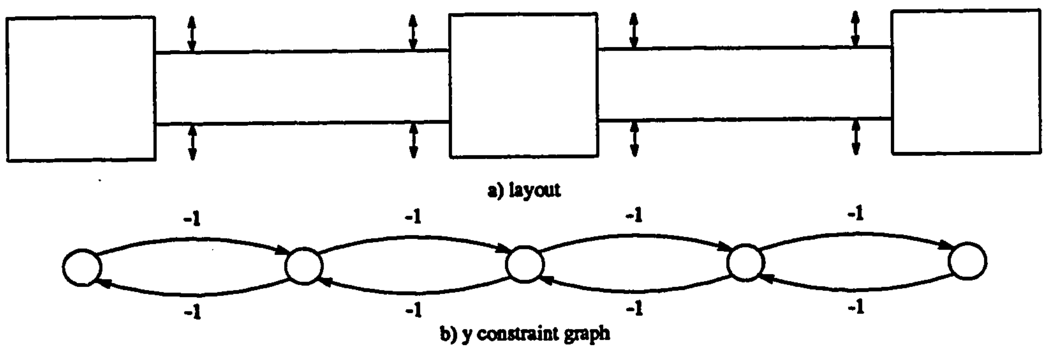


Figure 5.3 Cause of large strong components

analogous to strong components. Thus, it appears that decomposition is only useful when there are no spring arcs. Since all of the realizer problems use L^1 minimization, DSC decomposition is never any more useful than SCC decomposition and will not be discussed further. L^1 minimization forces realization problems to be solved without the benefit of a decomposition technique.

How does SCC size vary with problem size? If the average SCC size were constant, then the solution time would be linear in the size of the problem, whereas if the number of SCCs were constant, then the total solution time would be proportional to average solution time of the SCCs, which is probably nonlinear in its size. The data indicates that the size of the SCCs does in fact grow with problem size, but only very slowly. The results of performing a linear regression (number of vertices is the independent variable) are as follows. The best-fit lines and exponentials for the average SCC sizes of all problems without L^1 minimization are:

TC	$1.68+.00299*\nu$	$.80\nu^{0.21}$
FP	$1.29+.00213*\nu$	$.85\nu^{0.13}$

The average SCC size is small, and grows slowly. However, because the time required to solve a single strong component is clearly non-linear in the number of vertices, the above analysis understates the effect of large strong components. Because the time to solve a single SCC is potentially worse than linear in its size, the root-mean-square component size may be a more relevant measure, as it gives a greater weight to larger components:

TC	$1.65+.0155*\nu$	$.55\nu^{0.37}$
FP	$1.55+.00363*\nu$	$.80\nu^{0.19}$

Again, the growth with ν is slow (but not quite as slow). In all the above regressions, the data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

were quite noisy. The number of vertices, therefore, is only a weak predictor of strong component size.

By using SCC decomposition to divide a longest-path problem and solve each component in topological sort order, the time required to solve longest-path problems can be made closer to linear in the number of edges. SCC decomposition was not implemented for use in Lava because of the small number of problems where it was applicable and because the performance of the solvers without SCC decomposition was quite good (as will be discussed in §5.3.)

In summarizing, most problems are small; the number of vertices is approximately proportional to area; most problems are relatively sparse except for the second step of stretch-rule generation. The problems fall into two categories with respect to decomposition: those that because of L^1 minimization cannot be usefully decomposed, and those that can be. Of the decomposable problems, most all decompose into many small components via SCC decomposition.

5.2. Solution Methods

In this section I will discuss a collection of solution methods for the optimization problems previously described. These methods vary as to the type of problems solved, their ability to utilize (or or their need for) residual information remaining from a previous compaction step, their ability to utilize special structure of the problem, and in their relative speed. The solution methods are *incremental closure* (constraint propagation), *initial feasible solver* (IFS), and *network flow* (NF).

In the last chapter, I described how the symbolic layout translation problem can be compiled into a system of linear inequalities (constraints), which may also be represented by a directed graph. This system of inequalities is then solved to minimize the difference between a variable representing a boundary of the cell, and variable representing positions of objects within the cell. (The equivalent operation on the graph is computing the longest paths from the boundary vertex to each of the other vertices.) This system may be solved by NF, or if a feasible solution to the system exists, by IFS. If wire-length minimization is to be performed, NF must be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

used. The tiling algorithm used for compilation of the Abut cell type requires adding constraints to a closed system of constraints, for which the incremental closure algorithm is used.

Incremental closure is only useful for adding a constraint to a closed[8] system of constraints and maintaining the closure. While the initial closed system of constraints could be constructed by this method, there are far more efficient methods. However, it does not handle L^1 minimization (spring arcs). The initial feasible solver, which is a variation on an algorithm by D. B. Johnson [Johnson 73], will efficiently compute the longest path from a vertex provided that a feasible, though not necessarily optimal, solution is provided. This algorithm is useful when the solution from the previous compaction step is retained, or for computing longest paths from multiple source vertices. It does not handle L^1 minimization. Network flow (NF) is a special case of linear programming where the constraint equations each have a single +1 and a single −1 coefficient. NF is useful for computing the longest path. Although L^1 minimization is not a linear programming problem, by introducing artificial variables and some additional constraints, it is possible to transform it into a linear programming that can be solved by network flow.

Incremental Closure

The incremental closure algorithm is used for adding a constraint to an already closed system of inequalities and maintaining the closed property. The system of inequalities is represented as an adjacency matrix A: If there exists an inequality $x_i+C \le x_j$ then $a_{ij}=C$; else, if there does not exist an inequality between x_i and x_j, then $a_{ij}=-\infty$.

The algorithm is as follows:

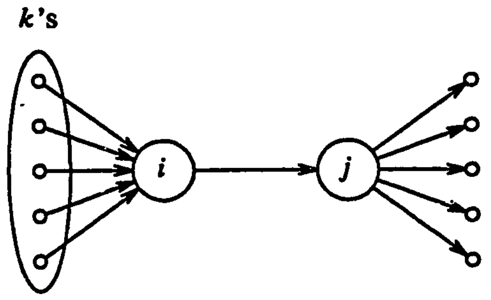
[8] In a closed system of constraints, if there is an inequality $x_i+A \le x_j$, and an inequality $x_j+B \le x_k$, then there will be an inequality $x_i+C \le x_k$, where $C \ge A+B$ and C will be the minimum necessary to satisfy this property.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

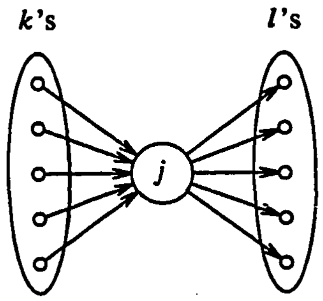
```
Input:   An additional arc from $i$ to $j$ with an arc length $C$ to be included in the system.

Step 1:  $a_{ij} \leftarrow \max(a_{ij}, C)$

Step 2:  Foreach $k$ $a_{kj} \leftarrow \max(a_{kj}, a_{ki} + a_{ij})$

Step 3:  Foreach $k, l$ $a_{kl} \leftarrow max(a_{kl}, a_{kj} + a_{jl})$
```

The algorithm works as follows:

In step 1, the new constraint from i to j is added. A now represents the longest path from all vertices to i, from i to j, and from j to all vertices. In step 2, the longest paths from all vertices (k) to vertex j through vertex i are computed. See figure 5.4. A now represents the longest paths from all vertices to j and from j to all vertices. In step 3, the longest paths from all vertices (k), through vertex j, to all vertices (l) are computed. Note that k and l both independently range over the all vertices. A now represents the longest paths from all vertices to all vertices, which means that we have computed the closure.



a) New arc (i, j) added prior to step 2



b) After step 2. All paths from k's through i to j have been computed

Figure 5.4 Constraint propagation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The algorithm is obviously $O(v^2)$. In the designs for which the optimization problems were measured, the algorithm did not contribute sufficiently to the overall CPU time (.01%) to make further analysis of the problems worthwhile.

Initial Feasible Solver

If a feasible solution is known, we can make use of it to solve the longest-path problem by a fast and simple algorithm. This algorithm is a variation of a portion of D. B. Johnson's algorithm AP [Johnson 73]. With the feasible solution we can process each edge only once by using a priority queue to determine the correct sequence.

The algorithm can be described by using a simple physical analogue. Consider a collection of rectangular blocks that we wish to pack together horizontally. The interaction of the blocks can be represented by constraints $x_i+s_i \le x_j$, where $\{x_i\}$ are the positions of the left edge and $\{s_i\}$ are the sizes of the blocks. The initial feasible solution is the initial positions of the blocks[9]. To perform this packing we place a long board vertically to the left of all the blocks and move it from left to right (its position is represented by x_{source}). Each time we contact a block, the block starts moving along with the board. At this point we can compute how far the block must move before it contacts each of its neighbor blocks to its right by examining each of the outedges from the corresponding vertex in the graph. This distance is the current amount of slack in the constraint. We then queue each of these blocks (vertices) on a priority queue based on the amount of this slack. If it is already on the queue, we may change its position. The order in which the blocks come off the queue is the order in which the blocks begin moving. In effect we are performing an event-driven simulation of the physical process.

The algorithm assumes that the initial feasible solution is in $\{x_i\}$ and that $x_{source}=0$:

[9] Feasible in this analogy means that the blocks are not overlapping.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for all $v$ $visited_v \leftarrow FALSE$
priority queue $\leftarrow (0, sourcevertex)$
while queue not empty {
   $(c, v) \leftarrow$ priority queue
   if not $visited_v$ { /* if we have not already processed this vertex */
      $visited_v \leftarrow$ TRUE
      $x_v \leftarrow x_v - c$
      for each edge $e : (v, u)$ {
         if not $visited_u$ { /* if already visited must have been a "back edge" */
            $k \leftarrow x_u - x_v + l(e)$
            if $k < 0$   failure – not feasible
            priority queue $\leftarrow (k, u)$
         }
      }
   }
}
```

If the priority queue operations take time proportional to the log of the size of the queue, then the worst case running time of this algorithm is $O(e \log v)$.

To use IFS to solve the multiple-longest-paths problem, network flow (*q.v.*) is used to find the longest paths from the vertex representing the left edge to all other vertices. (Due to the way the constraint graph is constructed, there will always be paths to all other vertices.) This solution is then always feasible no matter which vertex is used as the source vertex. IFS is then used to compute the longest paths from all other desired source vertices.

Network Flow

The longest-path problem can be cast as a network flow-problem. This section describes the network-flow problem, and how to transform the longest-path problem to it.

The network-flow problem (NF) is a special case of linear programming. The problem is:

$$\begin{array}{ll} \text{minimize} & c\,y \\ \text{such that} & A\,y = r \\ \text{and} & 0 \le y \le U, \end{array}$$

where, A is a vertex-edge incidence matrix[10] that specifies the graph; r is the vector of the number of units of flow that a vertex sources[11]. y is a vector representing the flows for each

[10] Each column (corresponding to an edge) contains a single +1 and a single−1 corresponding to the vertex that the edge comes from, and the vertex that it goes to, respectively.

[11] A sink vertex has a negative r_i, and $\sum r_i = 0$, *i.e.* the net flow into or out of the entire network must sum to zero.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

edge, which must be non-negative and less than the upper bounds U; and c is the cost per unit flow for each edge. (If A were unrestricted, this formulation would become the general linear programming problem.) Several methods for efficiently solving this problem are presented in [Kennington 80], but will not be discussed here; Lava uses the network simplex method.

The longest-path problem can be described in terms of a network-flow problem by using A to represent the graph, c to be the negative of the arc lengths, and forcing one unit of flow to go from the source vertex to the vertex to which the longest path is to be computed. For this problem, the upper bound on y is not used (U is infinite). The optimizer will choose a path for the one unit of flow of minimal cost (maximal sum of arc lengths).[12] The cost will be the negative of the path length (sum of arc lengths along the path), and the optimal cost will be the negative of the length of the longest path.

How do we compute the longest paths from the source vertex to each of the remaining vertices? Construct a network flow problem to find the minimal cost flows for a unit of flow from the source vertex to each of the other vertices. In this case the total cost is the sum of the longest paths and not very interesting. Assuming that the flows are along a spanning tree[13], the tree can be traversed in time $v-1$, computing the costs per unit flow to each vertex. Most of the solution algorithms compute and maintain the marginal cost information as a by product of their operation. These costs are known as the *dual variables*.

In the above, the longest-path problem is being solved by solving the dual problem. The dual of the network flow problem (without the upper bounds U) is [14]:

$$\begin{array}{ll} \text{maximize} & rx \\ \text{such that} & xA \le C \end{array}$$

[12] I have assumed that the optimal flow will be along a single path. In this case the assumption is correct. The assumption, however, is irrelevant as the costs, which will be minimal, are what is of interest.

[13] This assumption is not really necessary. If there are two equal cost paths, splitting the flow in any manner between the paths would yield an optimal solution, although most solution algorithms will choose either extreme. Since the costs (which are what count here) are the same, a tree is unnecessary except for the sake of explanation.

[14] see [Dantzig 63] for computing duals

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the single-pair, longest-path problem this dual becomes

$$\begin{array}{ll} \text{maximize} & x_{source}-x_{sink} \\ \text{such that} & x_i-C_{ij}\le x_j \end{array}$$

which is the longest-path problem.

For the construction of the network-flow problem from the longest-path problem given above, the problem in one domain being infeasible corresponds to the optimal solution of the dual being unbounded. If the L.P. problem has a loop of positive weight, the minimal network flow cost will be $-\infty$ with an infinite flow. If there are unreachable vertices in the longest path problem, the N.F. will not have feasible flows, and the longest path to those vertices will be $-\infty$.

Some network-flow algorithms need to start with an initial feasible solution. The usual method is to add some artificial variables (edges) with the initial feasible flow in the edges, but with a very high cost, so that the artificial edges will be quickly eliminated from the problem if it is feasible. For solving the longest path, artificial arcs are unnecessary because an initial feasible solution can be determined using a depth-first search of the graph.

Unfortunately, in practical applications, an algorithm that solves a correct (feasible) L.P. problem is not enough. If the problem is infeasible, we need to be able to diagnose the cause of infeasibility and determine which vertices and edges are part of the loop in order to produce diagnostic information meaningful to the IC designer. Also, producing a partially correct solution with the loop excised can be very useful for diagnosis.

Assuming that the L.P. problem is feasible, the maximal flow on any edge of the N.F. problem is $v-1$. Setting an upper bound of $v-1$ on all the flows will break the loop, but makes detecting the existence of the loop difficult, as a feasible problem may also have a flow of $v-1$. An upper bound of v will identify at least one edge (the flow will be v) in the loop and produce a partially correct solution. An upper bound of at least $2v-1$ will identify all edges in all (disjoint) infeasible loops (the flows will be $\ge v$) while producing a solution satisfying all but a few of the edges contained in the loops.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

L^1 minimization

As discussed in the previous chapter, it is very useful to be able to optimize

$$min \sum W_k \left| x_{i_k} - x_{j_k} + C_k \right|$$

over the system of inequalities in order to minimize wire lengths. This problem is no longer a longest-path problem, nor is it even a linear programming problem: the objective in non-linear. The section describes how it can be converted into a linear programming problem, and more specifically, a linear programming problem of the form which may be solved by the network simplex method used in the previous section[15].

In order to transform $\sum W_k \left| x_{i_k} - x_{j_k} + C_k \right|$ into a linear objective function, some artificial variables are added. For every term in the sum, define $\alpha_k - \beta_k = x_{i_k} - x_{j_k} + C_k$. The function $\sum W_k(\alpha_k + \beta_k)$ is minimized subject to $\alpha_k \geq 0$ and $\beta_k \geq 0$ and all the constraints on the x's.

Theorem. Minimizing $|x_1 - x_2 + C|$ subject to linear inequalities is equivalent to minimizing $\alpha + \beta$ such that $\alpha, \beta \geq 0$, where $x_1 - x_2 + C = \alpha - \beta$.

All the linear inequalities relevant to x_1 and x_2 can be summarized as

$$c_1 \leq x_1 - x_2 + C \leq c_2$$

which is equivalent to

$$c_1 \leq \alpha - \beta \leq c_2 \qquad \text{(i)}$$

At the minimum of $\alpha + \beta$, one of the α or β must be zero (otherwise α and β could be reduced by the lesser and still satisfy (i)). If $\alpha\beta = 0$, then $\alpha + \beta = |\alpha - \beta|$. Therefore, at the minimum of $\alpha + \beta$, $\alpha + \beta = |x_1 - x_2 + C|$.

There is one more artificial variable than necessary, and it can be eliminated. By substituting:

$$\alpha = \beta + C + x_1 - x_2$$

the objective becomes:

[15] The general simplex method is comparatively inefficient for the type of problem as it requires $O(ve)$ storage and $O(ve)$ time for each pivot. Though the number of pivots theoretically can be exponential, it is typically linear in v.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\text{minimize} \quad 2\beta + x_1 - x_2 + C$$
$$\text{such that} \quad \beta \ge 0,\ \beta + x_1 - x_2 + C \ge 0$$

This formulation is now a linear programming problem, although it does not appear to be of a form that can be solved by the more efficient network algorithms previously discussed. The network-flow problem admitted arbitrary linear objective functions, but required that the columns of the matrix (constraints) contain a single +1 *and*–1. Rewriting the problem:

$$\begin{bmatrix} X & B \end{bmatrix} \begin{bmatrix} A & S \\ 0 & -I \end{bmatrix} \le \begin{bmatrix} C_c & C_s \end{bmatrix}$$
$$B \ge 0$$
$$\textit{minimize} \begin{bmatrix} X & B \end{bmatrix} \begin{bmatrix} -r \\ 2 \end{bmatrix}$$

where A is the +1,–1 column matrix representing the constraint graph, S is a matrix of similar form representing the spring arcs, B is a vector of βs, and r is a vector containing the difference of the number of spring arcs entering and leaving each vertex (*i.e.*, it is the row sum of S). The dual of this problem is

$$\textit{minimize} \begin{bmatrix} C_c & C_s \end{bmatrix} \begin{bmatrix} y \end{bmatrix}$$

subject to

$$\begin{bmatrix} A & S \\ 0 & I \end{bmatrix} \begin{bmatrix} y_c \\ y_s \end{bmatrix} \begin{matrix} = \\ \le \end{matrix} \begin{bmatrix} r \\ 2 \end{bmatrix}$$
$$y_c \ge 0,\ y_s \ge 0$$

or

$$\begin{bmatrix} A & S \end{bmatrix} \begin{bmatrix} y \end{bmatrix} = r$$
$$0 \le \begin{bmatrix} y \end{bmatrix} \le U,$$
$$\textit{where } U = \begin{bmatrix} \infty \\ 2 \end{bmatrix}$$

which is exactly the form of the network flow problem.

Several algorithms are used for solving the systems of constraints Lava uses to translate symbolic layouts. If wire-length minimization is not required, and a feasible solution is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

available, a fast algorithm is available to take advantage of this information. Otherwise, the much more general network flow formulation is used and solved using the network simplex method. With appropriate problem formulation, network flow is also able to minimize wire lengths as its objective, or diagnose infeasible constraint systems. For tiling abut cell types an $O(v^2)$ constraint propagation algorithm is employed. In the next section the practical CPU time performance of these algorithms is measured and analyzed.

5.3. Performance Results

For many of the algorithms of the previous section, the theoretical asymptotic time complexity is only known for the worst case, but not for the typical case, or is otherwise not useful for predicting performance. The asymptotic time complexity may not be useful because it depends on measurements of the structure of the problem that have little meaning outside the algorithm and are therefore not very informative, or because there are constants such that the asymptotic behavior is not relevant for the size of problems encountered. For these reasons, an empirical study of the CPU time as a function of problem size was undertaken for the solution methods of the previous section.

Regression analysis was used to estimate how the CPU time depends on both the number of edges and the number of vertices.

$$t=k_1 e^{k_2} \qquad (e = \text{number of edges})$$

and

$$t=k_1 v^{k_2}$$

The data were broken down according to problem type. The types are longest path, L^1 minimization, and longest path from multiple source vertices[16]. For the longest path and L^1 minimization, the network flow (NF) algorithm was used. For multiple longest path (MLP), NF was used to find the first solution, and the IFS algorithm was used for all subsequent solutions. The breakdown according to problem type is different than in §5.1, where the breakdown was by

[16] If all vertices were used as source vertices, this would be the all-pairs longest-path problem.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

problem source, because a given constraint solver may be used for problems from a variety of sources. The solution algorithms are being studied in this section, rather than the problems. Problem types I (initial feasible generation) and A (Abut cells) only use longest path. Problem type C (compaction) uses either longest path or L^1, depending upon whether wire length minimization is required, and then uses MLP for the first step of abstraction generation. The second step of abstraction generation (problem type S) also uses MLP. The scatter plots of the data are shown in figures 5.5a through 5.5h.

The parameters computed for the models are as follows:

CPU time as function of edges		$k_1e^{k_2}$		
Layout	Problem Type	k_1	k_2	fit
TC	LP	.22	1.05	.957
	L^1	.28	1.14	.978
	MLP	.29	1.07	.953
	MLP per source	.089	.93	.967
FP	LP	.24	.98	.975
	L^1	.20	1.20	.986
	MLP	.12	1.32	.978
	MLP per source	.113	.85	.989

Times are in milliseconds on a VAX/780.

The solution times for the longest path problem are very close to linear in the number of edges. This result is pleasing because NF is a variant of the simplex method, which has an exponential worst case time dependence on problem size. L^1 minimization, a more difficult problem, is slightly slower than linear in e.

The time for MLP solution is worse than linear in the number of edges: exponents range from 1.07 to 1.32. There are several reasons for this behavior. The number of source vertices grows with problem size, making larger problems more complex in a way that is not measured just by probiem's size in edges. In other words, the constraint system is not simply bigger and more complex, but we are also asking for additional results to be produced (specifically $v \cdot s$ numbers, where s is the number of source vertices). Regression analysis to predict the number of source vertices (from the total number of vertices) shows a poor fit to the data, although it is clear that it is increasing with the problem size, but not quite as fast as linear in the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

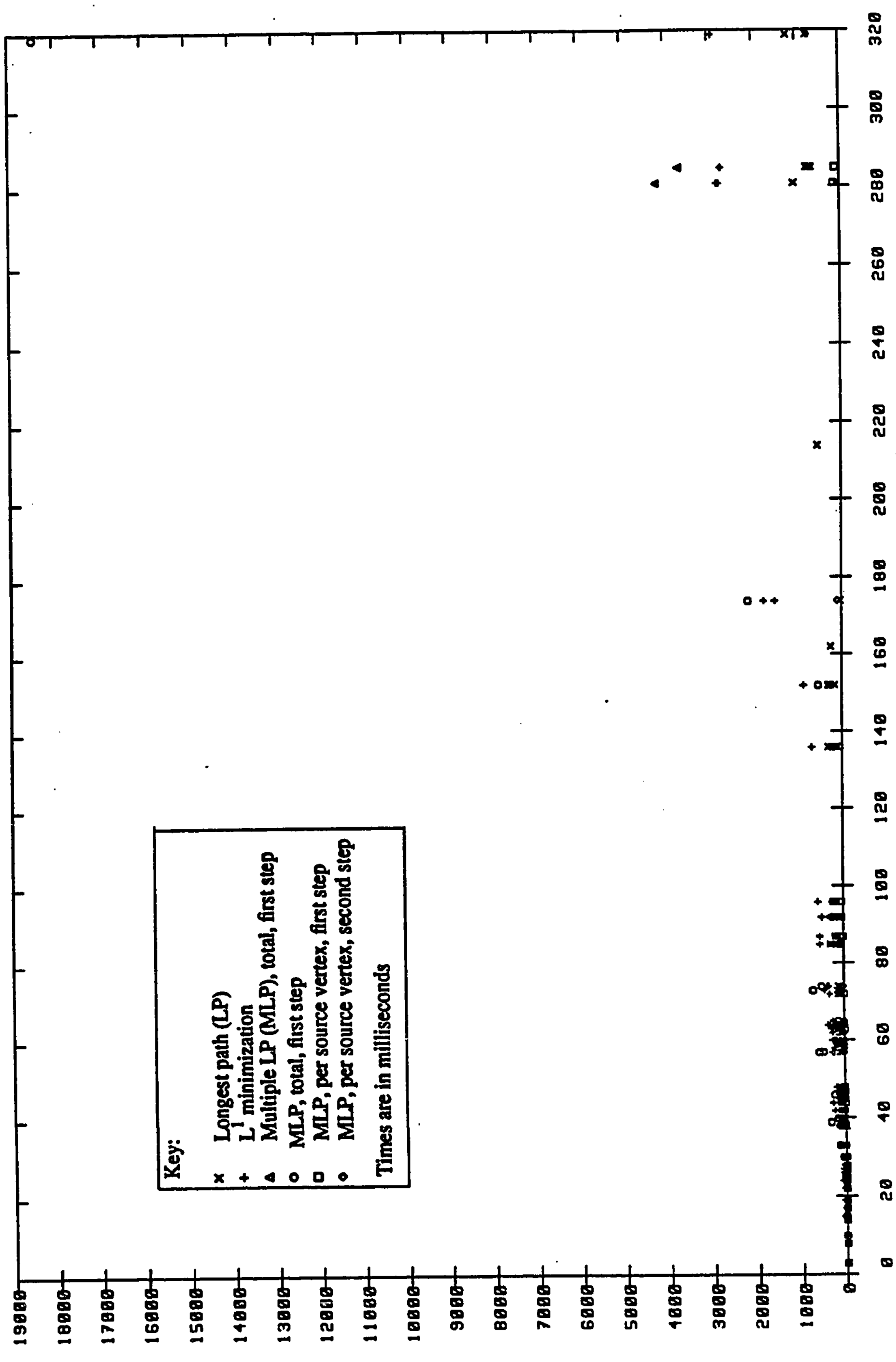


Figure 5.5a Time vs. vertices for "TC" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

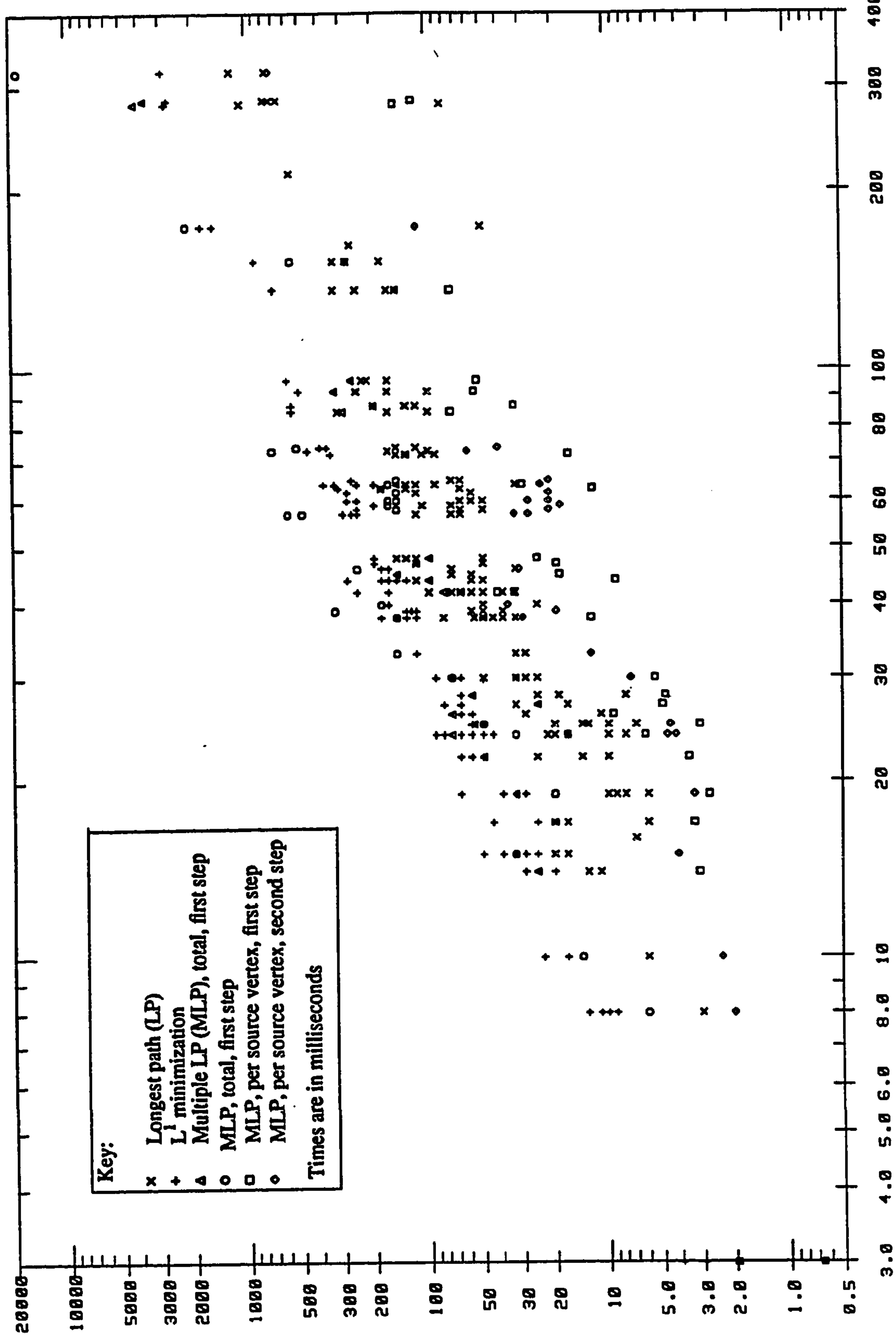


Figure 5.5b Time vs. vertices for "TC" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

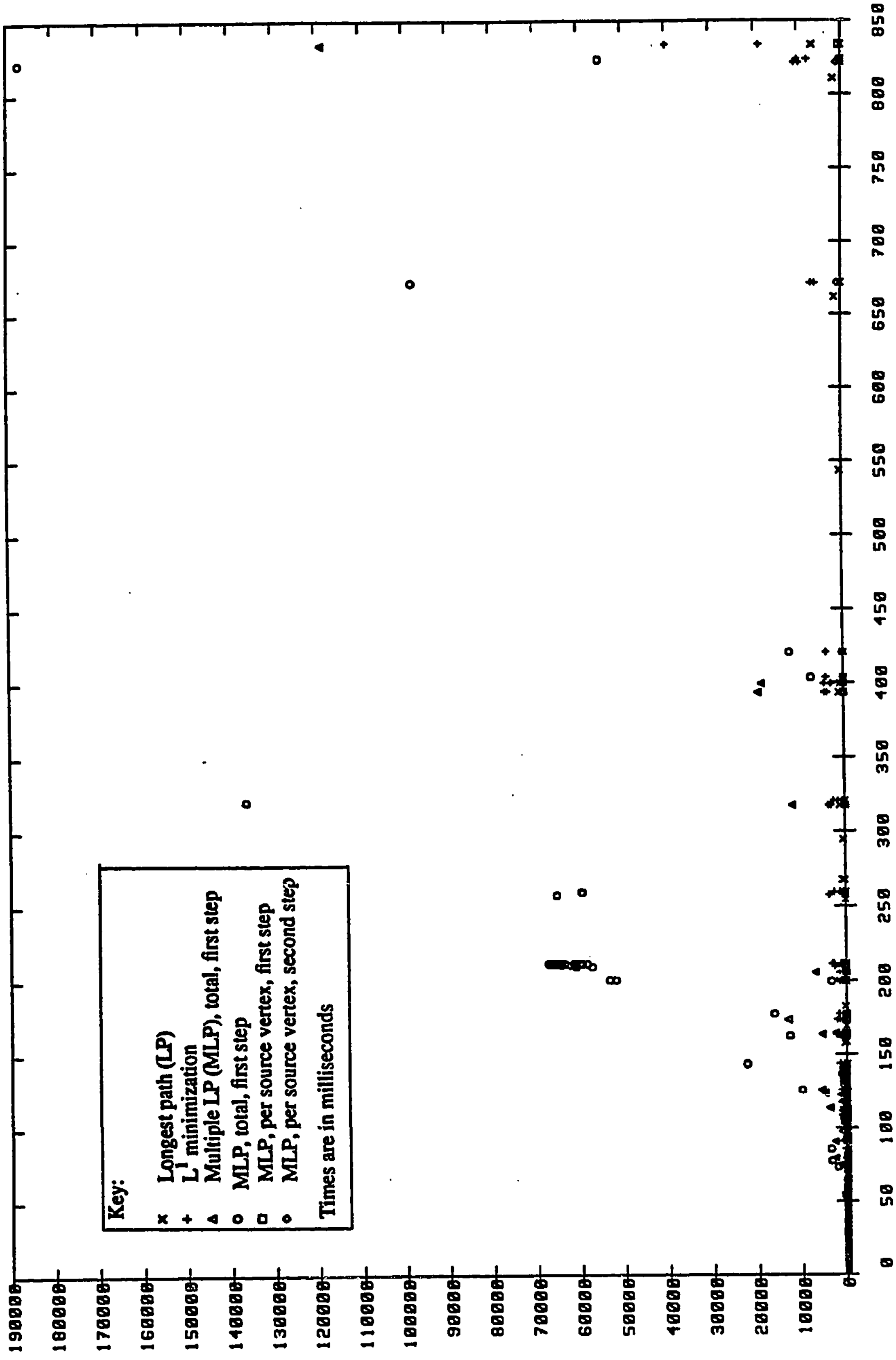


Figure 5.5c Time vs. vertices for "FP" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

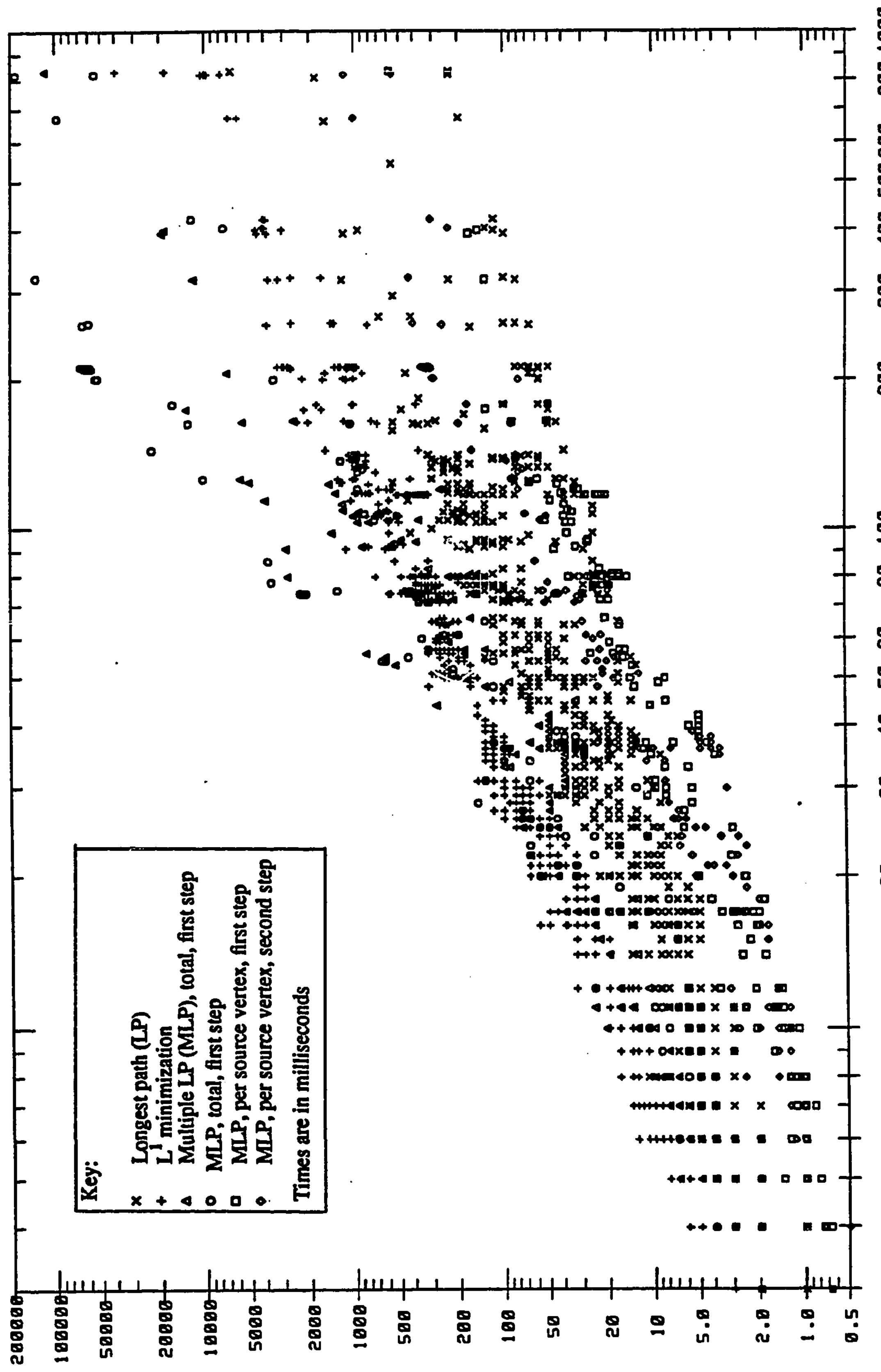


Figure 5.5d Time vs. vertices for "FP" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

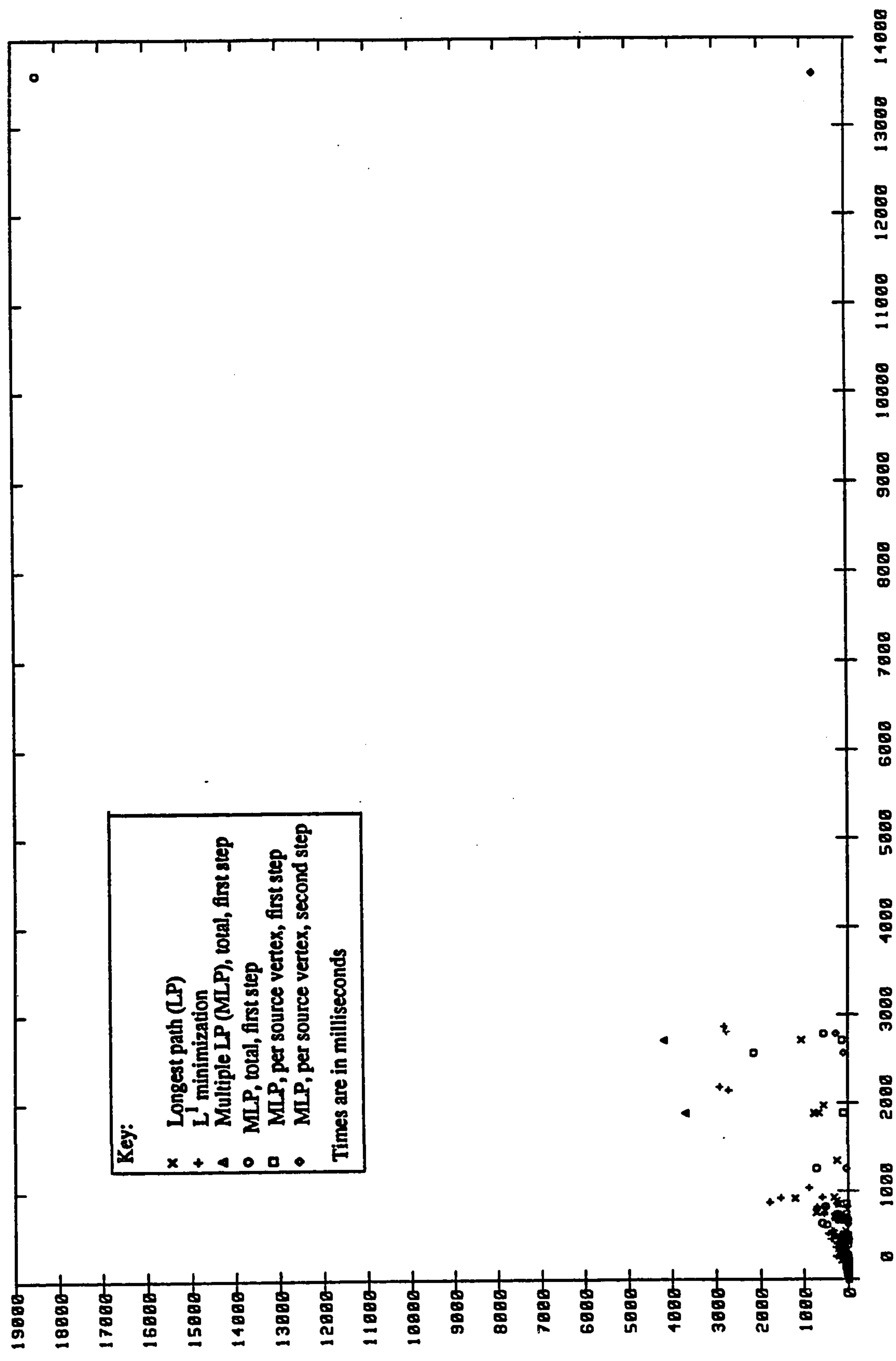


Figure 5.5e Time vs. edges for "TC" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

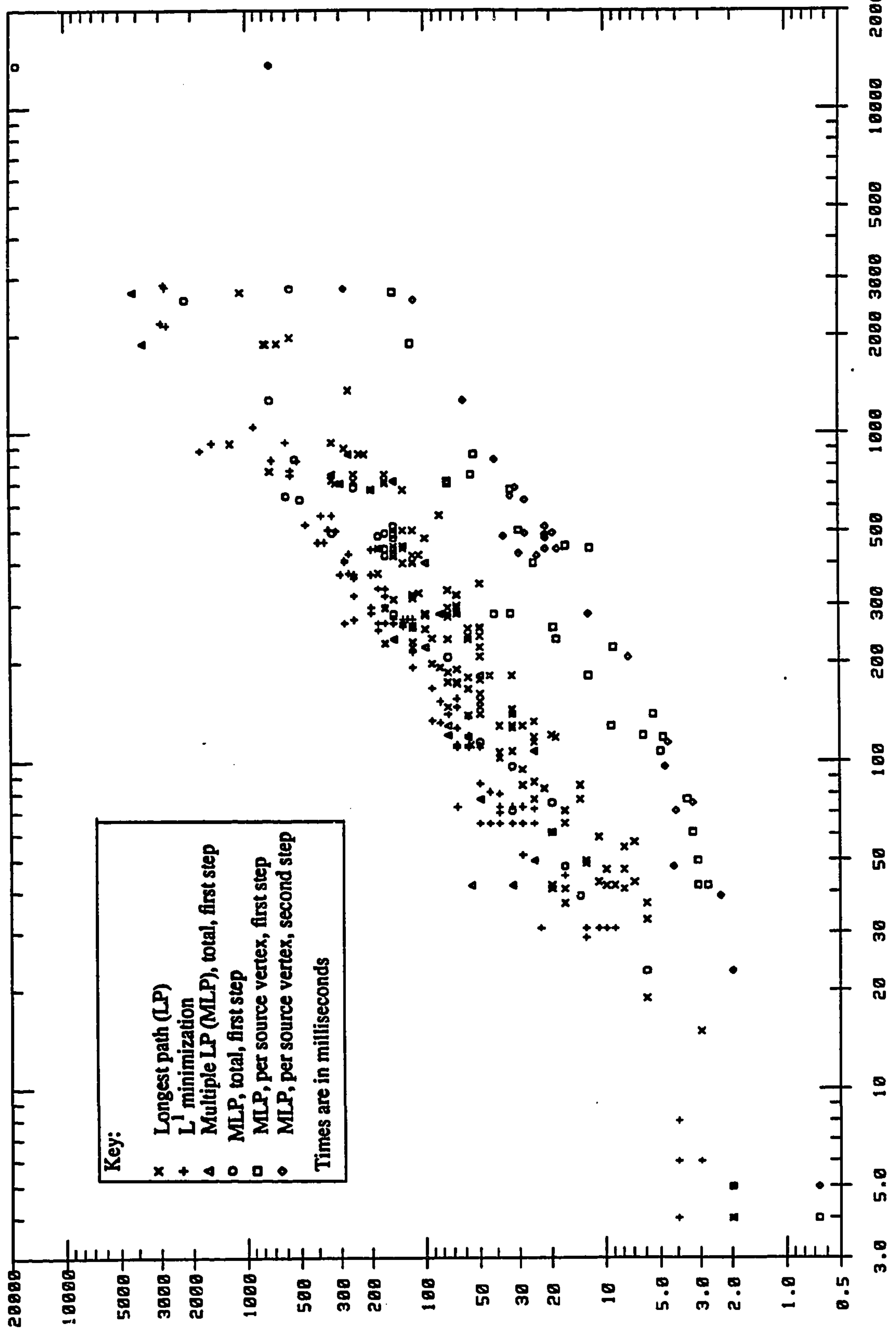


Figure 5.5f Time vs. edges for "TC" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

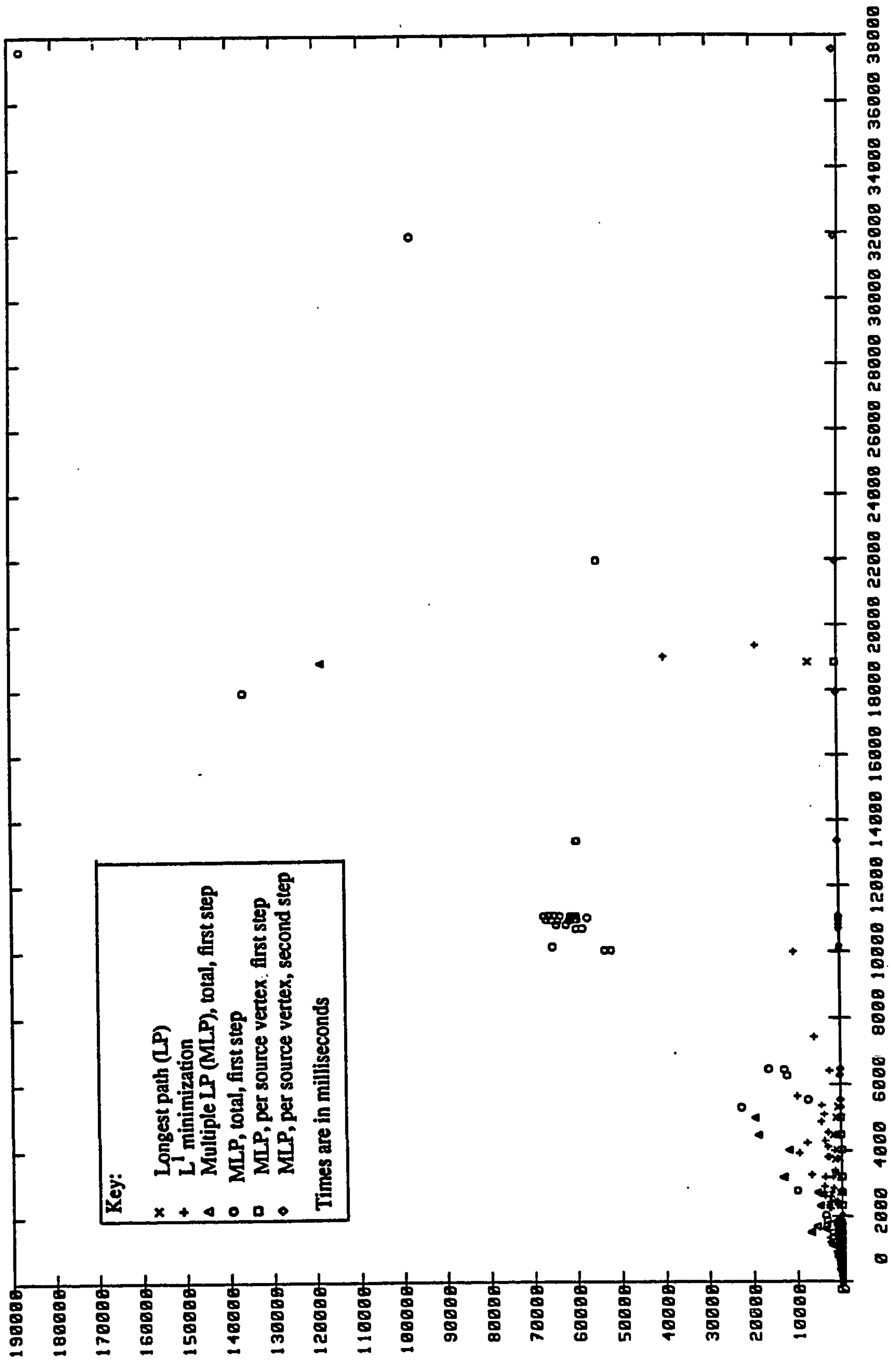


Figure 5.5g Time vs. edges for "FP" (linear scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

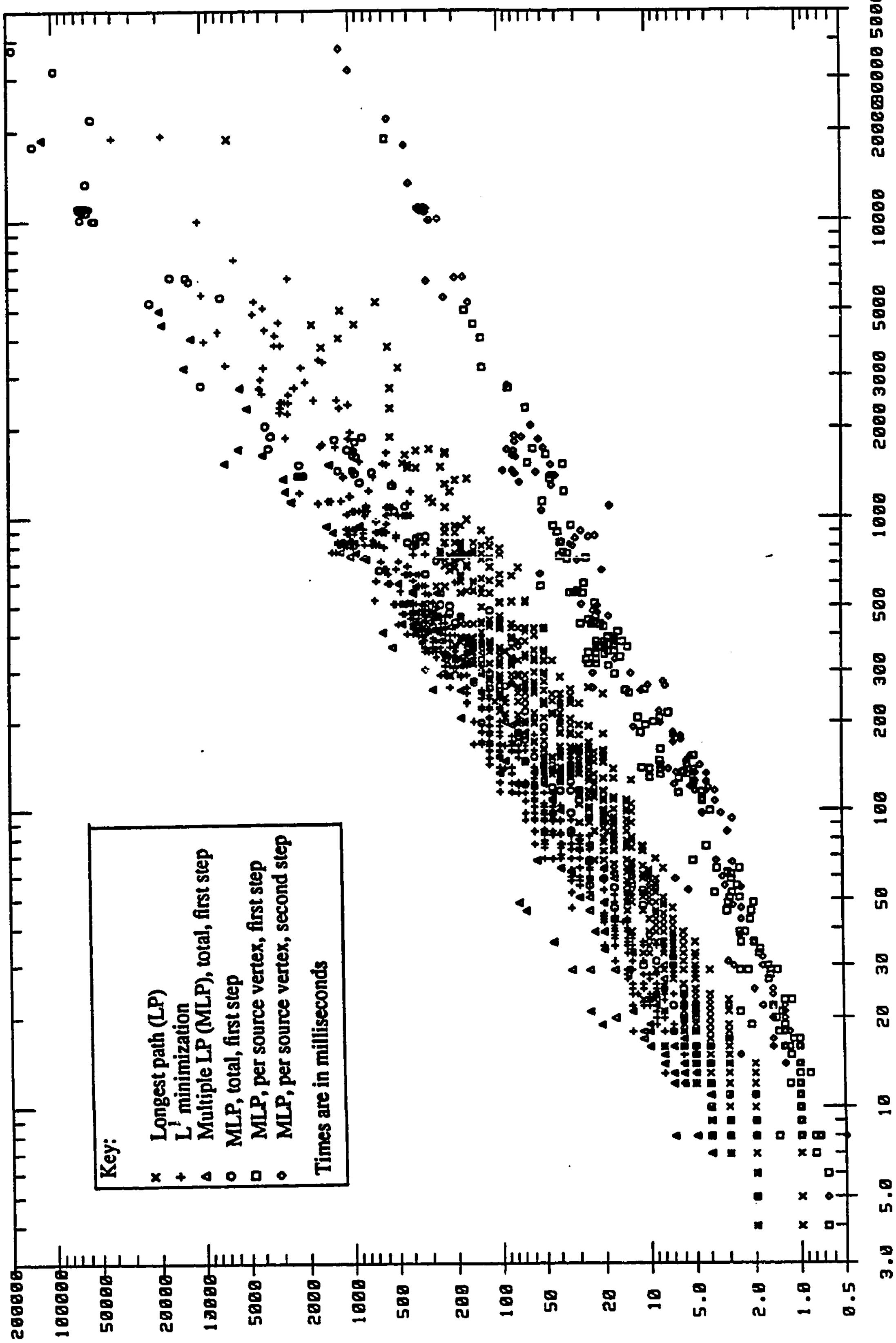


Figure 5.5h Time vs. edges for "FP" (log scale)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertices[17]. Here, the data are also broken down as to first and second step of stretch-rule generation. As was discussed in §5.1, the second step of stretch rule generation uses a much denser constraint graph:

	Number of Source Vertices		
	All (fit)	First Step (fit)	Second Step (fit)
TC	$3.1v^{0.24}$ (.086)	$3.7v^{0.17}$ (.112)	$2.5v^{0.31}$ (.058)
FP	$.93v^{0.72}$ (.075)	$2.0v^{0.46}$ (.066)	$.52v^{0.91}$ (.062)

A more appropriate measure of the performance of the solver might be the time per source vertex[18]. The time per source vertex is approximately linear in the number of edges. Theoretically, the time per source vertex is bounded by $O(e \log_2 v)$[19] (see discussion of IFS algorithm).

When solution time s examined as a function of the number of vertices, which is more closely related to the cell's complexity, it is less attractive, but in most cases, still quite good.

CPU time as function of vertices				
Layout	Problem Type	$k_1 v^{k_2}$		
		k_1	k_2	fit
TC	LP	.25	1.40	.921
	L^1	.50	1.53	.988
	MLP	.23	1.65	.931
	MLP per source	.076	1.42	.937
FP	LP	.37	1.16	.920
	L^1	.49	1.51	.988
	MLP	.061	2.12	.925
	MLP per source	.066	1.40	.961

The time for the longest-path problem as a function of the number of vertices is one of the few measurements that do not show good consistency between the two chips: the exponent ranges form 1.40 to 1.16. When viewed this way, the CPU time is worse, but, with the exception of MLP, not that bad ($<v^{1.5}$).

[17] The number of source vertices is approximately the number of terminals that a cell has, while the number of vertices is closely related to the number of components in the cell. In this light, a poor correspondence seems reasonable. Also, by assuming a constant number of components per unit area and a constant number of terminals per cell boundary length, one could argue that the number of terminals should be proportional to the square root of the number of components.

[18] The times reported as time per source vertex are exactly that. When the times required to compute the first row (which is done by a different method) is subtracted off, and the time per source vertex not including the first is computed, the data in most cases showed a poorer fit and never better.

[19] It is not surprising to not see the effect of the $\log_2 v$ factor in comparison to e, especially when half of the MLP problems are problem source "S" (second step stretch rule generation; the densest one).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The MLP time per source vertex is somewhat disappointing. To determine the reason, I have split the data into the first and second steps of stretch-rule generation. The graphs for the second step are denser, as the constraint generation must be more conservative[20].

			$k_1e^{k_2}$			$k_1v^{k_2}$	
		k_1	k_2	fit	k_1	k_2	fit
TC	MLP Step1	.433	1.00	.919	.331	1.51	.948
	MLP Step1/vert	.087	.94	.951	.089	1.34	.931
	MLP Step2	.212	1.12	.977	.161	1.82	.941
	MLP Step2/vert	.084	.92	.984	.064	1.51	.953
FP	MLP Step1	.216	1.20	.963	.208	1.68	.937
	MLP Step1/vert	.113	.86	.987	.102	1.21	.976
	MLP Step2	.076	1.39	.986	.027	2.44	.948
	MLP Step2/vert	.110	.85	.989	.051	1.53	.973

From this data, one can see that the problem is due to the much denser graph in the second step: the exponent on the time as function of the number of edges does not vary much from first step to second step, but varies significant between steps on the time as a function of the number of vertices. The time per source vertex is always approximate linear in the size of the input data (number of edges).

What can be done to improve this second step time? There appears to be three approaches:

1) Reduce the amount of data (number of edges) that must be examined for each source vertex by either performing some reduction on the graph, or by processing several source vertices simultaneously. Neither approach appears very promising, although they may be topics for future research.

2) Improve the constraint generation. As discussed in Chapter 4, more information about physical connectivity could be used in order to allow the shadowing algorithm to be used for this second step constraint generation, thereby producing a sparse graph. Presumably, if the topology is used, the number of other objects against which a constraint must be written would be small and either constant or slowly growing with the cell's size, even for the second-step stretch rules, thereby resulting in e being close to linear in v.

[20] First and second step of stretch rule generation correspond to problem sources called "C" and "S", respectively, in §5.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3) If the source of these problems is examined (see table below), it will become apparent that the worst of them come from Abut cell types. As mentioned in Chapter 4, the abstractions for the abut cell types could be produced directly, without the sticks constraint-generation algorithms, and without the concern for the x/y interlock which requires the conservative constraint generation and produces the dense graphs. The fact that they are not done that way was simply a matter of programming expediency.

MLP problems							
	Data Points	Avg. Verts.	Max. Verts.	Avg. Edges	Max. Edges	Avg. Time (ms)	Total Time (ms)
TC abut	4*	127	281	1383	2720	1619	6475
TC non-abut	52	57.35	319	684.67	13604	581.3	30227
FP abut	122	157.14	834	4457.13	37598	20622.6	2515959
FP non-abut	264	32.22	163	231.39	1870	142.3	37562

* With only four data points (two abut cells) in TC, only the data from FP is significant.

The following table gives total cpu time (for optimization problem solution) and number of problems for each problem type, with and without adjusting for the unnecessary abut cell type computations (as mentioned above):

CPU time totals					
	Prob. type	Number of Problems		CPU time (seconds)	
TC	LP	145	43%	16.5	19%
	L^1	136	40	34.0	39
	MLP	56	17	36.7	42
TC (adjusted)	LP	137	42	15.2	20
	L^1	134	41	29.4	39
	MLP	52	16	30.2	40
FP	LP	992	39	67.3	2
	L^1	1150	45	428.9	14
	MLP	386	15	2553.5	84
FP (adjusted)	LP	809	39	41.6	12
	L^1	1028	49	278.1	78
	MLP	264	13	37.6	11

Removing the spurious abut problems reduces the CPU time spent solving the optimization problems for FP from 51 minutes to 6 minutes. An additional benefit not shown in the above data, is a corresponding significant reduction in constraint generation time.

The total CPU time Lava used to compile the layouts (without the Abut cell improvements) is given in the following table, along with percentages spent on various tasks[21]:

[21] Some discussion of these percentages is in order. The total solving times don't match the preceding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	Time	Parsing	Constraint[22] Generation	Solving	Other Overhead
TC	951	5.2%	60.2%	18.1%	16.5%
FP	12777	8.2	39.8	36.2	15.8%

Note that as a result of analysis and improvements of both the constraint generation and solution algorithms, solving the optimization problems is now a modest percentage of the total CPU time.

In summary, the solution time for most problems is not much worse than linear in the size of the problem (in number of edges), although most of the CPU time is used for the solution of a few large problems that have dense constraint graphs. These large problems come from the last step of abstraction generation on large cells. By changing the way Abut cells are processed these problems can be avoided, resulting in an eightfold reduction in CPU time used for constraint solving.

Through analysis of the constraint solving problems, and of the performance of the constraint solving algorithms, I have determined which portions of Lava are dominant in their CPU usage and which one will be dominant as the chip size increases. Methods of avoiding these dominant problems have been proposed, and the effects of one of these solutions calculated. The remaining problems appear to be mostly between $O(v)$ and $O(v^{1.5})$, where v is closely related to the area.

"CPU time totals" table because the development of the running Lava system lags behind the development of constraint solving algorithms. The principle difference is that the running Lava uses an older algorithm for MLP which is about 33% - 46% slower than the one described above.

[22] The constraint generation algorithm used in this execution was an experimental implementation which was slower in many cases, but was asymptotically better, and produced fewer constraints. In a practical system, the constraint generation algorithm would be chosen on the basis of cell size, but such algorithm switching would make such asymptotic studies difficult at best. Note that the percentage of CPU time spent doing constraint generation is much less in "FP" than in "TC", mainly due to the greater number of large cells and the fact that constraint generation is asymptoticly better than solving.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. Conclusion

Custom ICs have potentially great performance and economic benefits over other methods of IC design, but also have much higher costs for design and verification. A significant and tedious portion of the custom IC design process is layout, which is the task of translating the electrical circuit to geometric description while obeying the design rules and minimizing the size. Symbolic layout promises to simplify the layout task, but previously has not been practical for an entire chip for several reasons. Among the reasons are: the symbolic layout translation program takes too much CPU time for large designs; the translator puts too many restrictions on the layouts that it can produce; and the quality of the output is poor. My goal was to determine how to produce a practical symbolic layout tool.

To achieve this goal, a symbolic IC layout language, Lava, was designed to capture the designers intent in a design-rule-independent manner. Lava allows the designer to represent his design at a high level of abstraction with minimal sacrifice of flexibility or silicon area.

Methods of compiling Lava that use minimal CPU time and result in high-quality layouts were developed. First the description is compiled to an intermediate form consisting of lists of components and interconnections. This intermediate form could as well be generated from a graphical IC description method.

The intermediate form is then compiled into a series of optimization problems in such a manner that the problems are relatively small and easy to solve, yet with minimal sacrifice of quality of the resulting layout. In order to keep the problems small, the design is broken into pieces according to the hierarchy provided by the designer: an *abstraction*, giving the external characteristics of a cell, is created for each cell. The abstraction contains the rules by which the external connection points may move relative to each other and still allow the construction of a design-rule-correct cell. A new method is used to construct this abstraction so that motion of connection points in one dimension does not invalidate the rules for the other dimension. Also, a *shadowing* algorithm is employed for constraint generation to minimize the number of constraints. In order to maintain the quality of output, the constraint graph can have cycles, allowing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

flexible components and connections. A weighted sum of absolute values of wire lengths is minimized. This L^1 wire-length minimization problem, which is not a linear optimization problem, is transformed by introducing additional variables into a linear programming problem which may be efficiently solved using a network flow algorithm.

In order to chose or develop appropriate optimization algorithms, it was necessary to analyze and characterize the problems. Most problems are small and sparse. Decomposition was studied, but for a variety of reasons was not useful. In order to gauge the performance of the optimization algorithms, an empirical study of CPU time as a function of problem size was undertaken. The optimization algorithms were usually slightly worse than linear in the number of constraints. Most of the CPU usage was due to a certain class of problems from the Abut cell type. Making changes to the way the Abut cell type was handled allowed an eightfold improvement in the total CPU time required by the optimization algorithms.

I have demonstrated that a practical symbolic layout translator that produces a high-quality layout in a modest amount of CPU time is possible.

What can be done to improve the speed? The solution time of the optimization problems cannot be improved much further, since it is already not much worse than linear in the size of the problem (size of the input data) in most cases. To reduce the amount of time spent solving the problems, the size or number of optimization problems solved must be reduced.

To reduce the size and number of optimization problems, unnecessary work can be cut down in a number of ways: 1) Better constraint generation algorithms — the use of circuit topology in constraint generation should allow optimization problems with a number of edges which is linear in the number of components (even in the second step abstraction problems). The current constraint generation method treats the cells as collections of disconnected and unrelated rectangles. 2) As mentioned earlier, composition cells (Abut) can be compiled with far less work. 3) Sometimes it is not cost-efficient to compute abstractions, particularly in small cells, or cells which are used rarely. Some flattening of the hierarchy in selected cases could save CPU time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I expect that with the improvements mentioned here and others that symbolic layout translation will eventually be done in a time which is not much worse than linear in the size of chip, possibly $O(n \log n)$ or $O(n^{1.5})$.

Contributions

This dissertation provides several contributions to the field of symbolic IC layout translation. These contribution fall into two main categories: new or improved translation algorithms, and studies of optimization problem characteristics and optimizer performance.

New methods were necessary to improve the speed of symbolic layout translation and to improve the quality of the output. Among them are the methods of generating stretch rules and abstractions in such a manner that the x and y problems are decoupled, thereby permitting hierarchy in the design to be exploited in a fail-safe manner. Another method to improve speed is the "Shadowing" method of constraint generation which generates a sparse (and therefore easier to solve) system of constraints. In order to improve the quality of output, L^1 minimization was used to reduce wire length, and a method of converting the L^1 minimization problem into the much more tractable network flow problem was developed.

In order to chose or design the most appropriate optimization algorithms for the problems created during translation, both the characteristics of the optimization problems, and the performance of the solution algorithms, were measured. While the characterizations of the optimization problems and optimizer performance may be affected somewhat by artifacts of the symbolic layout system, I believe that they are typical of what occurs in many different symbolic layout systems. Finally, I have demonstrated that practical symbolic layout can be CPU efficient.

While there is undoubtedly more than one way to produce a practical symbolic layout translator, I have presented a set of design decisions which successfully leads to a practical translator. The measurements which I made allow Lava to be used as a standard of comparison when exploring some of the alternatives.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix

Of the several layouts which Lava was used to design, there were two on which optimization data were collected. One layout is a tester controller chip ("TC") with 1633 transistors (including PLA) and the other is the data path section from a floating point processor chip ("FP") with 3499 Transistors. The plots of these layouts are shown in Figures A.1 and A.2. The following table summarizes some other information on the complexity of the layouts:

		Layout	
		TC	FP
Transistors		1633	3499
	Lines	5k	18k
Source Code	Bytes	118k	376k
	Files	37	171
Size of Layout		$3.17M\lambda^2$	$4.07M\lambda^2$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

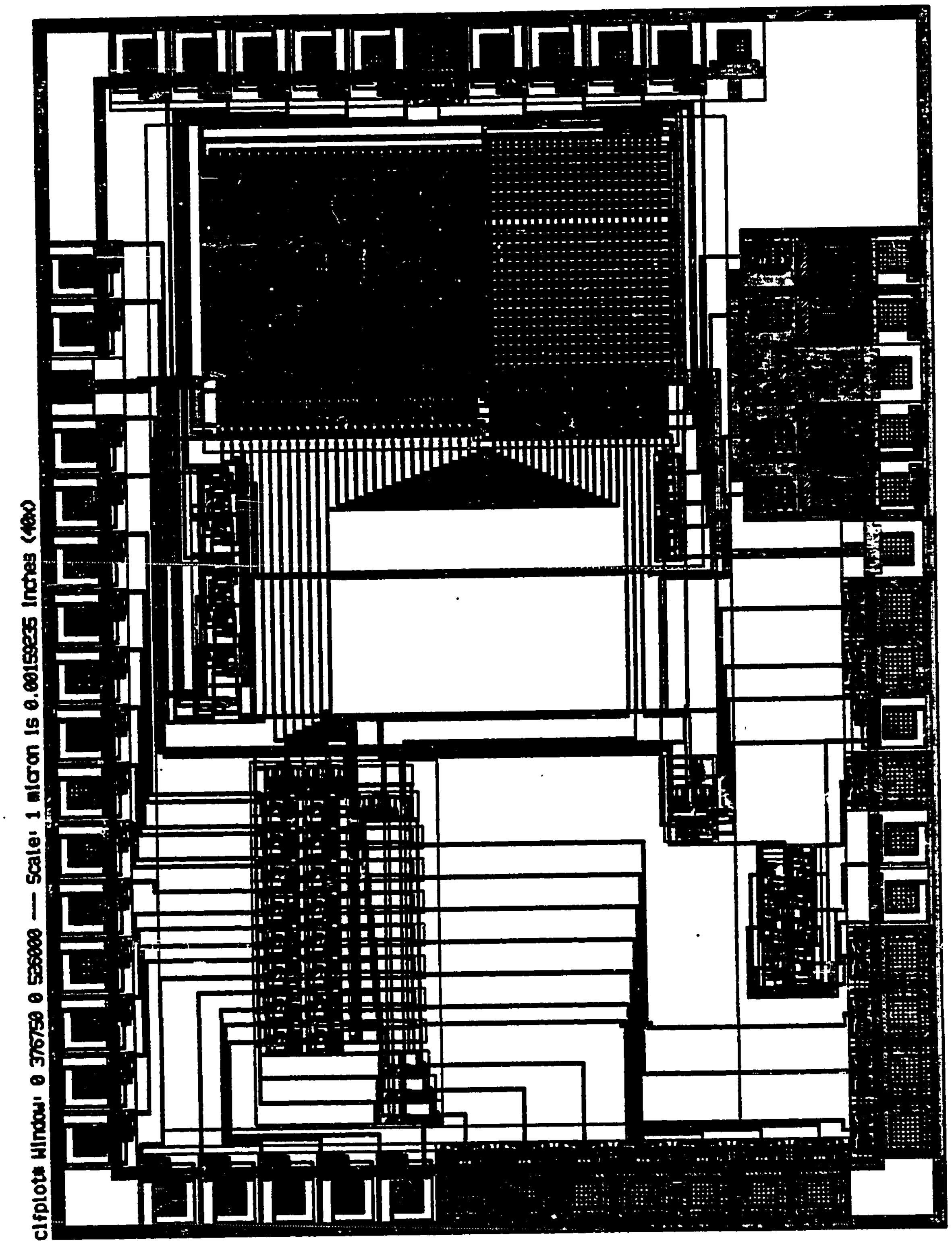

clfplot: Window: 0 376750 0 536000 — Scale: 1 micron is 0.0015925 inches (40x)

Figure A.1 Tester controller chip ("TC")

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

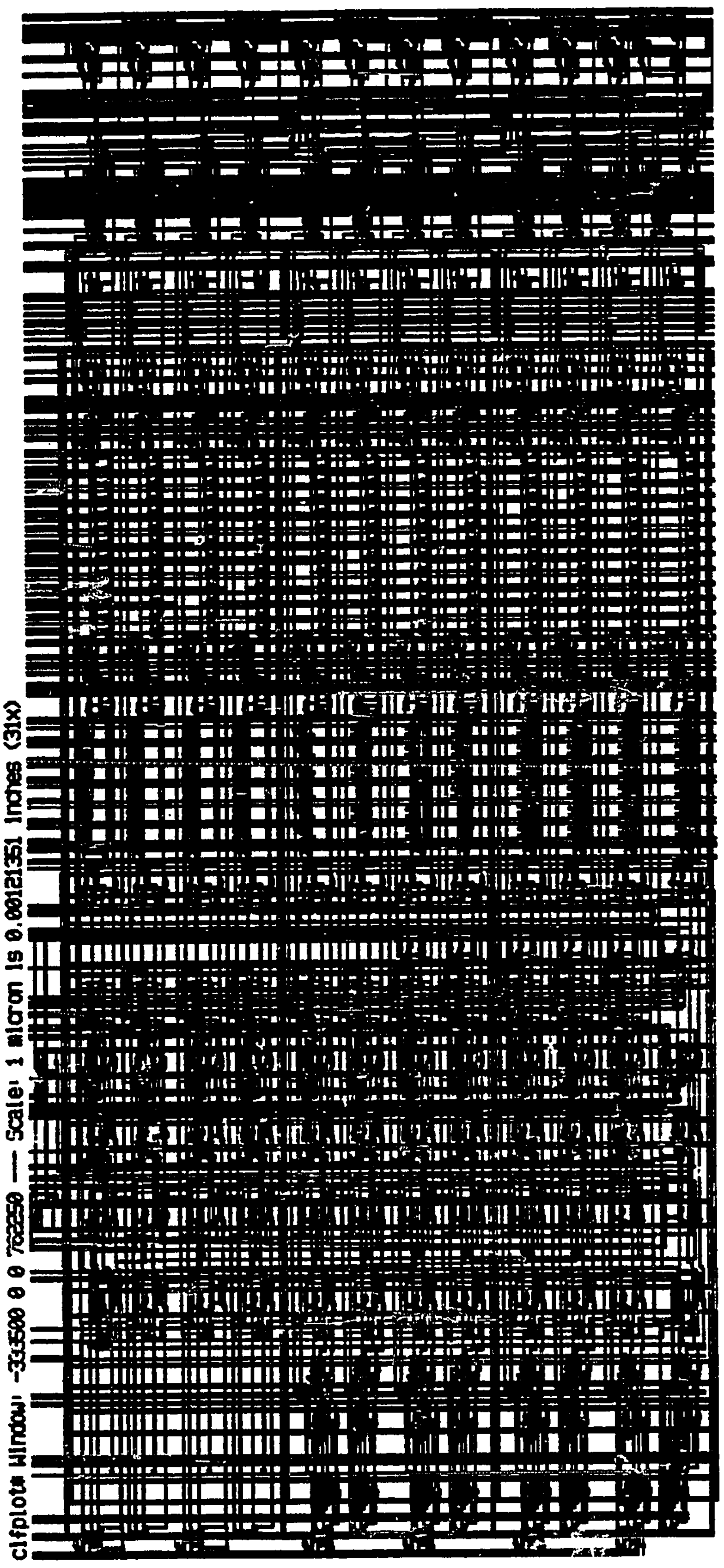

Cifplot* Window: -333500 0 0 762250 —— Scale: 1 micron is 0.00121351 inches (31x)

Figure A.2 Floating point data path layout ("FP")

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[Aho 74] A. Aho, J. Hopcroft, and J. Ullman.
The design and Analysis of Computer Alorithms.
Addison-Wesley, 1974.

[Akers 70] Sheldon B. Akers, James M. Geyer and Donald L. Roberts.
Ic Mask Layout with a Single Conductor Layer.
In *Seventh Annual Design Automation Workshop*, pages 7-16. ACM/IEEE, 1970.

[Barnes 75] Donald E. Barnes and Thomas L. Davis.
Graphics Layout Systems for Faster Designs.
In *Wescon Professional Program.* Western Electronic Show and Convention, September 1975.

[Boyer 83] David G. Boyer and Neil Weste.
Virtual Grid Compaction Using the Most Recent Layers Algorithm.
In *IEEE International Conference on Computer-Aided Design*, pages 92-93. IEEE, September 1983.

[Buric 83] M. R. Buric, *et. al.*
The Plex Project: VLSI Layouts of Microcomputers Generated by a Computer Program.
In *IEEE International Conference on Computer-Aided Design*, pages 49-50. IEEE, September 1983.

[Burns 82] Charlie Burns and Tim Saxe.
Sedit Users' Manual.
Technical Report, Stanford University, 1982.

[Dantzig 63] George B. Dantzig.
Linear Programming and Extensions.
Princeton University Press, 1963.

[Dunlop 78] A. E. Dunlop.
SLIP: Symbolic Layout of Integrated Circuits with Compaction.
Computer Aided Design 10(6):387-391, November 1978

[Dunlop 81] A. E. Dunlop
SLIM–The Translation of Symbolic Layouts into Mask Data.
Journal of Digital Systems V(4):429-451, 1981.

[Eichenberger 82] Robert Mathews, John Newkirk, and Peter Eichenberger.
A Target Language for Silicon Compilers.
In *Compcon Proceedings*, pages 349-353. IEEE Computer Society, Spring 1982.

[Floyd 62] Robert Floyd.
Algorithm 97, Shortest Path.
Communications of the ACM 5(345), 1962.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Gibson 76]	Dave Gibson and Scott Nance. SLIC–Symbolic Layout of Integrated Circuits. In *Thirteenth Design Automation Conference*, pages 434-441. ACM/IEEE, June 1976.
[Hsueh 79]	Min-Yu Hsueh. *Symbolic Layout and Compaction of Integrated Circuits.* PhD thesis, University of California at Berkeley, December, 1979.
[Johnson 73]	D. B. Johnson. *Algorithms for Shortest Paths.* PhD thesis, Cornell University, 1973.
[Juran 83]	B. Juran. A Totally Symbolic Approach to VLSI Layout. In *IEEE International Conference on Computer-Aided Design*, pages 53-54. IEEE, September 1983.
[Kedem 83]	Gershon Kedem and Hiroyuki Watanabe. Graph-Optimization Techniques for IC Layout and Compaction. In *Twentieth Annual Design Automation Conference*, pages 113-120. ACM/IEEE, 1983.
[Kedem 84]	Gershon Kedem and Hiroyuki Watanabe. Graph-Optimization Techniques for IC Layout and Compaction. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems* CAD-3(1):12-20, 1984.
[Kennington 80]	J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming.* John Wiley & Sons, 1980.
[Lengauer 79]	T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. In *ACM Transactions of Programming Languages and Systems 1(1):121-141, July 1979.*
[Liao 83]	Y. Liao and C. K. Wong. An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints. In *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems, CAD-2(2):62-69, 1983.*
[Lipton 82]	R. J. Lipton *et. al.* ALI: a Procedural Language to Describe VLSI Layouts. In *Nineteenth Design Automation Conference*, pages 467-474. IEEE 1982.
[Mathews 81]	Robert Mathews. Unpublished algorithms for constraint solving. 1981.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[McGarity 83] Ralph C. McGarity and Daniel P. Siewiorek.
Experiments with the SLIM Circuit Compactor.
In *Twentieth Design Automation Conference*, pages 740-746. ACM/IEEE, June, 1983.

[Mosteller 81] R. C. Mosteller
REST: A leaf Cell Design System.
In *VLSI 81: very large scale integration*, Academic Press, 1981, pages 163-172.

[Rosenberg 84] Jonathan B. Rosenberg.
Chip Assembly Techniques for Custom IC design in a Symbolic Virtual-Grid Environment.
In *Conference on Advanced Research in VLSI*, pages 213-225. Massachusetts Institute of Technology, January, 1984.

[Sastry 82] S. Sastry and A. Parker.
The Complexity of Two-Dimensional Compaction of VLSI Layouts.
In *International Conference on Circuits and Computers*, pages 402-406. IEEE, September 1982.

[Schiele 83] W. L. Schiele.
Improved Compaction by Minimized Length of Wires.
In *Twentieth Annual Design Automation Conference*, pages 121-127. ACM/IEEE, 1983.

[Tarjan 81a] R. E. Tarjan.
Fast Algorithms for Solving Path Problems.
In *Journal of the ACM* 28(3):594-614, July 1981.

[Tarjan 81b] R. E. Tarjan.
A Unified Approach to Path Problems.
In *Journal of the ACM* 28(3):577-593, July 1981.

[Tarjan 74] R. E. Tarjan.
Testing Flow Graph Reducibility.
In *Journal of Computer and System Sciences* 9:355-365, 1974.

[Weste 81] Neil Weste.
Virtual Grid Symbolic Layout.
In *Eighteenth Annual Design Automation Conference*, pages 225-233. ACM/IEEE, 1981.

[Williams 78] John D. Williams.
STICKS-A Graphical Compiler for High Level LSI Design.
In *AFIPS Conference Proceedings*, pages 289-295. AFIPS, 1978.

[Wolf] W. H. Wolf.
Two-dimensional Compaction Strategies.
PhD Thesis, Stanford University, 1984.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.