Efficient Interconnection Schemes for VLSI and Parallel Computation

by

Ronald I. Greenberg

B.S. Systems Science and Mathematics
B.S. Computer Science
A.B. Mathematics
M.S. Systems Science and Mathematics
Washington University, St. Louis
(1983)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy in
Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
September 1989

Signature of Author____
Department of Electrical Engineering and Computer Science
August, 1989

Certified by____
Charles E. Leiserson
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Efficient Interconnection Schemes for VLSI and Parallel Computation

by

Ronald I. Greenberg

Submitted on August 11, 1989 to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
in Electrical Engineering and Computer Science

Abstract

This thesis is primarily concerned with two problems of interconnecting components in VLSI technologies. In the first case, the goal is to construct efficient interconnection networks for general-purpose parallel computers. The second problem is a more specialized problem in the design of VLSI chips, namely multilayer channel routing. In addition, a final part of this thesis provides lower bounds on the area required for VLSI implementations of finite-state machines.

This thesis shows that networks based on Leiserson's *fat-tree* architecture are nearly as good as any network built in a comparable amount of physical space. It shows that these "universal" networks can efficiently simulate competing networks by means of an appropriate correspondence between network components and efficient algorithms for routing messages on the universal network. In particular, a universal network of area A can simulate competing networks with $O(\lg^3 A)$ slowdown (in bit-times), using a very simple randomized routing algorithm and simple network components. Alternatively, a packet routing scheme of Leighton, Maggs, and Rao can be used in conjunction with more sophisticated switching components to achieve $O(\lg^2 A)$ slowdown.

Several other important aspects of universality are also discussed. It is shown that universal networks can be constructed in area linear in the number of processors, so that there is no need to restrict the density of processors in competing networks. Also results are presented for comparisons between networks of different size or with processors of different sizes (as determined by the amount of attached memory). Of particular interest is the fact that a universal network built from sufficiently small processors can simulate (with the slowdown already quoted) any competing network of comparable size regardless of the size of processors in the competing network. In addition, many of the results given do not require the usual assumption of unit wire delay. Finally, though most of the discussion is in the two-dimensional world, the results are shown to apply in three dimensions by way of a simple demonstration of general results on graph layout in three dimensions.

The second main problem considered in this thesis is channel routing when many layers of interconnect are available, a scenario that is becoming more and more meaningful as chip fabrication technologies advance. This thesis describes a system MulCh for multilayer channel routing which extends the Chameleon system developed at U. C. Berkeley. Like Chameleon, MulCh divides a multilayer problem into essentially independent subproblems of at most three layers, but unlike Chameleon, MulCh considers the possibility of using partitions comprised of a single layer instead of only partitions of two or three layers. Experimental results show that MulCh often performs better than Chameleon in terms of channel width, total net length, and number of vias. In addition to a description of MulCh as implemented, this thesis provides improved algorithms for subtasks performed by MulCh, thereby indicating potential improvements in the speed and performance of multilayer channel routing. In particular, a linear time algorithm is given for determining the minimum width required for a single-layer channel routing problem, and an algorithm is given for maintaining the density of a collection of nets in logarithmic time per net insertion.

The last part of this thesis shows that straightforward techniques for implementing finite-state machines are optimal in the worst case. Specifically, for any s and k, there is a deterministic finite-state machine with s states and k symbols such that *any* layout algorithm requires $\Omega(ks \lg s)$ area to lay out its realization. For nondeterministic machines, there is an analogous lower bound of $\Omega(ks^2)$ area.

Key words: interconnection networks, routing networks, parallel computation, supercomputers, universality, fat-trees, routing algorithms, randomized routing, multilayer channel routing, single-layer routing, VLSI, layout algorithms, finite-state machines.

Thesis Supervisor: Charles E. Leiserson
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would like to thank first my thesis adviser, Charles Leiserson, for his extensive guidance throughout my stay as a graduate student at MIT. Charles has taught me a great deal technically in addition to providing much professional advice. His diligent proofreading and insightful comments have also contributed greatly to the presentation of this thesis.

Many parts of this thesis represent joint work with other researchers. Chapters 3 and 5 are mostly derived from joint papers with Charles Leiserson, while most of Chapter 6 is derived from a joint paper with Alex Ishii of MIT and Professor Alberto Sangiovanni-Vincentelli of U. C. Berkeley. Section 7.1 represents joint work with Miller Maley of Princeton U. Chapter 8 is derived from a joint paper with Mike Foster of Columbia U.

I have also benefited from the help of many others at MIT. Among those who contributed to useful technical discussions are Tom Cormen, Tom Leighton, Bruce Maggs, and Miller Maley. Thanks also to Tom Leighton and Tom Knight for serving on my thesis committee. I also could not have completed my work as effectively without the help of those who maintained the local computing environment, especially Sally Bemus and Ray Hirschfeld.

Finally, I am grateful to those who have provided financial support for my graduate study. Especially helpful was a fellowship awarded by the Fannie and John Hertz Foundation. In addition to financial support, the directors of the foundation always showed a deep concern for the fellowship holders and a desire to help us complete our degrees as smoothly as possible. Additional support has been provided by the Defense Advanced Research Projects Agency under contract N00014–87–K–0825 and an earlier contract, and the Office of Naval Research under contract N00014–86–K–0593.

Contents

Chapter 1

Introduction

Space and time have long been key issues in VLSI design and parallel computation, just as they have been among the key issues throughout the history of computer science. Over and over, we ask how much hardware and how much time are required to perform various procedures. This thesis is concerned with space and time, with a particular emphasis on the efficient use of space in three contexts: the construction of general-purpose parallel computers, multilayer channel routing, and construction of finite-state machines.

Though the studies in this thesis share a strong concern for the efficient use of space, the contexts are only partly similar, and the techniques are quite different.

In the case of general-purpose parallel computation, the key concern is generality. Within the constraints of limited resources, it is desired to interconnect processors so that any communication pattern can be realized fairly efficiently. That is, given a fixed amount of space, we design a parallel computer which does not incur much penalty in execution time to simulate any other machine occupying comparable space.

In the case of channel routing, the goal is to obtain compact VLSI realizations for fixed interconnection patterns of pins in restricted locations. This problem may be relevant in the lower design levels of any computer, even a general-purpose parallel machine. There will always be fundamental operations which are best performed within a single processor and on a single chip. Providing the capability to perform these operations efficiently requires a specific set of interconnections, and certain design styles give rise to the problem of channel routing. It is only at higher levels, where it is not feasible to provide all the special-purpose connections which might be desired, that we seek a general set of connections which is almost as good as each special-purpose connection scheme.

The study of finite-state machines in the last part of this thesis deals with special-purpose machinery as does channel routing, but the perspective is one of realizing a functional specification rather than a physical interconnection pattern. Also, while the first two parts of the thesis emphasize the search for compact hardware layouts, the last part emphasizes lower bounds on space.

Despite the differences in the problems considered in this thesis, there is a continuous concern with the realization of devices in VLSI technologies. Underlying these studies is substantial commonality in the VLSI models and the broad goals as discussed in the next section.

1.1 VLSI Models

Space has already been highlighted as a dominant concern throughout this thesis. Time will also prove important, especially in the first part of this thesis. This section provides background on the structure and operation of VLSI chips which is necessary to develop appropriate models for the notions of space and time.

1.1.1 Space

By abstracting away some fabrication details, VLSI chips may be simply viewed as a patchwork of wires stacked in several layers. For the most part, these layers are insulated from one another so that the wires form independent conducting paths. But it is also possible to place holes (referred to as contact cuts or vias) in the insulation to connect wires in different layers. In addition, it is possible to fabricate a transistor at the intersection of wires in specific layers. From the fundamental components of transistors, which serve as switches, and the wires which serve as communication paths, complicated devices may be constructed. Though the number of layers varies from one technology to another, it may be treated as a fixed constant according to whichever technology has been selected. The key measure of chip size (and hardware cost) is then the area of the chip.

The practical realities of fabrication and the electrical properties of VLSI chips impose many limitations on chip design. A neat reduction of the requirements to a concise set of rules by which one could safely design chips (in a particular technology) was popularized by Mead and Conway [73]. Among the requirements are a minimum transistor size, minimum widths for wires of the various types, and minimum separations between wires.

For mathematical analyses of VLSI layout, it is desirable to adopt an even higher level of abstraction than a set of design rules in the style of Mead and Conway. One common simplification is to restrict wires to be rectilinear. This restriction often arises in practice, based on the language used to describe layouts or the fabrication techniques involved. Another common simplification is to dispense with the classification of wires into different types (according to their layer). That is, one value is assumed for minimum wire width and for minimum wire separation regardless of the actual types of the wires. Also, the dimensions of a via are taken to be equal to the minimum wire width. These simplifying assumptions are certainly reasonable for asymptotic analysis of wiring patterns since they change the required area by only a constant factor. Some of the assumptions may be more problematical in the context of optimizing a particular interconnection instance in a specific technology, as in channel routing. For simplicity, however, this thesis assumes rectilinear and "uniform" technology throughout, as is consistent with many standard design methodologies.

The assumption of rectilinear, uniform technology leads naturally to what may be referred to as "grid-based" models. Formal models of this type were first provided by Thompson [95, 96]. In his scheme, a VLSI chip is represented by a graph in which edges correspond to communication wires and nodes correspond to wire junctions, transistors, or other processing devices. The area required for such a graph is determined by obtaining an embedding in a two-dimensional grid which maps nodes to grid points and edges to edge-disjoint paths in the grid. (Path crossings may be implemented by using two different layers in the chip.) The correspondence between graphs and chips is maintained by restricting attention to bounded-degree graphs or by reserving extra space for processing elements corresponding to nodes of large degree [95]. In the context of asymptotic analysis, the model is adequate for VLSI technologies involving any constant number of layers by way of a correspondence scheme given by Thompson [96]. In the case of truly three-dimensional technologies, we may consider the obvious analog of embeddings into a three-dimensional grid. (In the latter work of Thompson [96], the notion of embeddings in a grid is replaced by tiling a plane with a finite set of tile types, but these two approaches are fundamentally equivalent.)

The modeling approach just outlined is used extensively in this thesis, but the concerns will shift slightly when we study multilayer channel routing. In the case of channel routing, we will be concerned with detailed routing of connections in the layers of a VLSI chip, and constants will not be brushed aside. Problem instances will consist of a set of terminals at specified points on two parallel lines. Each of these terminals belongs to a specific net, and we are required to route wires in such a way as to connect terminals belonging to the same net. These routings will consist of wire segments in the layers of a three-dimensional grid of height equal to the number of available routing layers along with vias which connect wire segments on adjacent layers.

1.1.2 Time

A popular and convenient assumption about time in VLSI circuitry is that exactly one bit can be transferred across a wire of any length in unit time. Some justification for this modeling assumption is provided in several sources [13, 15, 19, 95], and the assumption will generally be adopted in this thesis. There are, however, also good arguments for using a wire delay function which is more sensitive to wire length [25, 73, 88, 96]. For this reason, wire length issues will receive some attention in Chapter 4 within the first part of this thesis. Also, in the second part of this thesis, total wire length will be a secondary measure (after chip area) of the quality of channel routing solutions. In addition, timing considerations in a context where constant factors are important will motivate the inclusion of another secondary measure of the quality of channel routing solutions, namely total number of vias.

1.2 Thesis Overview

The first part of this thesis takes up the issue of connecting processors to form a general-purpose parallel computer. Though this question has been widely studied, most researchers have sought networks that can efficiently simulate other networks built from a comparable number of components. This thesis concentrates more on the use of physical space in accordance with the VLSI models outlined above. Chip area has long been recognized as a measure of VLSI cost, and usage of physical space is meaningful in the wider context of interconnecting the chips and boards comprising a parallel machine. As the number of processors in parallel machines grows into the millions, the difficulty of physically interconnecting them will hardly be an issue that can be ignored.

This thesis shows that several variants of Leiserson's fat-tree architecture [62] serve well as general-purpose parallel machines. These machines are "volume-universal", by which we mean that they can efficiently simulate any other machine of comparable volume. The notion of efficiency which we use is that the simulation should increase the computation time by only a small polylogarithmic factor of the volume used. Though volume-universality is of particular interest in our three-dimensional world, the mathematics and drawing of pictures are generally simpler in two dimensions, and, of course, basic chip fabrication technologies are two-dimensional. Thus, much of the discussion throughout this thesis will be in the two-dimensional context, but it will be shown that the results can be extended to three dimensions.

Chapter 2 discusses the notion of area-universality and introduces the fat-tree architecture. Fat-trees are based upon the structure of a complete-binary tree with processors at the leaves and switches at the internal nodes. But unlike the situation in a normal tree, which is "skinny all over", each *channel* between internal nodes of a fat-tree is a bundle of wires of appropriately chosen capacity. As outlined in Chapter 2, the procedure of showing fat-trees universal is twofold. First, the channel capacities must be chosen so that the fat-tree has a layout of modest area but can still support substantial communication bandwidth. That is, the channel capacities must be reasonably matched to the amount of communication induced by an appropriate mapping of competing networks to fat-tree components. Second, an efficient means of routing messages on the fat-tree must be demonstrated.

Chapter 3 considers the issue of efficiently routing messages on fat-trees. The context is that of *on-line* routing, as opposed to the *off-line* situation where the communication pattern is known in advance. The chapter provides randomized routing algorithms which yield good results even when the switches in the network are very simple. These algorithms work by randomizing in the choice of messages to send at any given time. All of these results are expressed in terms of a measure of congestion λ, referred to as the *load factor*, which is a lower bound on the time to deliver the messages. The load factor of a set of messages is the largest ratio over all channels in the fat-tree of the number of messages which must pass through the channel divided by the capacity of the channel. Chapter 3 presents first a general algorithm for routing on fat-trees and then an algorithm for an important special case of channel capacities. The number of delivery cycles (routing attempts) required on an n processor fat-tree is $O(\lambda + \lg n \lg\lg n)$ with high probability in the first case, and $O(\lambda)$ in the second case; each delivery cycle requires $O(\lg^2 n)$ time. Next an algorithm is presented for the variant fat-tree architecture referred to as the butterfly fat-tree, in which each switch has only a constant number of inputs and outputs. This algorithm uses $O(\lambda \lg^2 n)$ delivery cycles of time $O(\lg n)$. Note is also taken of faster algorithms requiring more complicated switches which have been suggested for the butterfly fat-tree by other researchers. In the remainder of the thesis, the routing details are generally abstracted away, and the simulation results are expressed in terms of the running time for the chosen message routing algorithm.

Chapter 4 completes the discussion of area-universal networks by detailing universality results in several contexts. Initially, the emphasis is placed entirely on communication issues by considering effective fat-tree channel capacities for comparisons against networks connecting the same set of processors. Later, the comparison is generalized to networks in which processors have different size as determined by the amount of attached memory. One example of the results shown is that an appropriate universal fat-tree of area $\Theta(A)$ built from processors of size $\lg A$ requires only $O(\lg^2 A)$ slowdown *in bit-times* to simulate any network of area A, without any restriction on processor size or number of processors in the competing network. Using the results of Chapter 3, it is possible to obtain $O(\lg^3 A)$ slowdown with simpler switches and a simpler algorithm. The universal network can also be designed so that any message traversing a path of length d in the competing network need follow a path of only $O(d + \lg A)$ length in the universal network. The bound on path length implies that in most cases, results based on the unit wire delay assumption apply equally well to wire delay functions that increase with distance. Finally, upper bounds are also given on the time required by a universal fat-tree to simulate parallel machines occupying greater space than the fat-tree.

Chapter 5 provides the foundations for extending the area-universality results to three-dimensions and volume-universality. This section provides a simpler demonstration of general 3D graph layout results, developed by Leighton and Rosenberg [57] and does a bit of generalization useful for the type of fat-tree structures considered in Chapter 4. These results give upper bounds on the area and maximum edge length required in laying out graphs, based on decomposition properties of the graph. The results are used to determine the layout volume for fat-trees with particular choices of channel capacities.

The second part of this thesis considers the problem of multilayer channel routing. Channel routing, which refers to the problem of realizing a fixed set of connections among pins that lie along two parallel lines, has been used to implement the layout of integrated circuits in a variety of design styles. The traditional channel routing problem assumes two layers of material are available for routing, but advances in manufacturing technology are making larger numbers of layers feasible. Thus, it is meaningful to extend channel routing to the multilayer case, and the ideas may also be applied to well-established technologies such as printed circuit boards and hybrid circuits, where several layers are used to implement interconnections.

Chapter 6 describes the system MulCh, which has been implemented to perform multilayer channel routing. MulCh is based on the Chameleon system [17], which divides a multilayer problem into essentially independent subproblems of two or three layers. MulCh differs from Chameleon in allowing also single-layer subproblems. Experimental results on realistic problems show that MulCh often performs better than Chameleon in terms of channel width, total net length, and number of vias.

Chapter 7 provides improved algorithms for certain subtasks performed by MulCh, yielding potential improvements in speed and performance. These opportunities for improvement are based on MulCh's "greedy" method of assigning nets to partitions. That is, nets are considered one at a time and assigned to that partition which appears to be the one which will have the smallest resulting channel separation. MulCh as implemented uses rough, easily computable estimates of required channel width, since this computation must be performed so frequently. But in the single-layer case, it is possible to determine exactly the minimum channel separation in time linear in the number of nets. The algorithm for doing this, described in Chapter 7, generalizes similar studies for the "river routing" problem, in which single-sided connections are disallowed. Due to the incremental nature of the procedure for building partitions, it would also be desirable to obtain algorithms which can compute estimates of channel separation more quickly by making use of previous information and simply updating the estimate in accordance with the addition of a single new net. One of the most useful estimates for channel width is a measure referred to as the density. included in Chapter 7 is an incremental algorithm for computing density in logarithmic time per net insertion.

The final part of this thesis, in Chapter 8, obtains lower bounds on the layout area of finite-state machines. It shows that straightforward layout techniques are optimal in the worst case. Specifically, for any s and k, there is a deterministic finite-state machine with s states and k symbols such that *any* layout algorithm requires $\Omega(ks \lg s)$ area to lay out its realization. Similarly, any layout algorithm requires $\Omega(ks^2)$ area in the worst case for nondeterministic finite-state machines with s states and k symbols.

Part I

General-Purpose Parallel Computation

Chapter 2

Area-Universal Routing Networks

In addition to formulating VLSI models as discussed in Chapter 1, Thompson laid the groundwork for a wide-ranging study of the area and time requirements of VLSI chips performing various computations [95, 96]. In fact, the issues of area and time have often been linked (as in Thompson's work), and many thorough analyses of the tradeoff between chip area and computation time have been performed. Thompson, himself, studied computation of the Discrete Fourier Transform and the sorting problem. Other examples of this type of study include works on basic arithmetic [1, 19, 22], matrix multiplication and related problems [87], and the class of *transitive* functions [108]. The above works, however, are relevant only to a particular problem or limited class of problems.

Other research works have sought more general results in the sense that they have provided computational structures employing limited resources which may be used to solve a wide generality of problems even though they cannot perform as well as the best special-purpose machine for every possible specialized problem. For example, Valiant [105] has shown that there is a combinational Boolean circuit of $O(s \lg s)$ gates that can be made to perform the same computation as any combinational Boolean circuit of s gates. In a different framework, several schemes have been demonstrated for efficiently routing permutations or random patterns of communications between processors linked together by "practical" networks [3, 4, 5, 6, 60, 77, 80, 84, 99, 100, 103, 104].[1] (The generally agreed upon notion of "efficient" is time polylogarithmic in the number n of processors.) Galil and Paul [41] carried this further by demonstrating, under various models of computation, a universal parallel machine which can efficiently simulate the behavior of any other network of the same number of processors. Other works have even shown that any parallel random-access machine (PRAM) can be efficiently simulated on a bounded-degree network [7, 8, 50, 74, 82, 101, 102, 107]. The strongest of these results [7, 50, 82] show that any PRAM computation completed in time t can be simulated in time $O(t \lg n)$ with high probability or $O(t \lg^2 n)$ deterministically.

The works just cited, however, have only a limited focus on the issue of space. They demonstrate devices which can simulate other machines of the same number of processors, using "practical" networks (usually equated with bounded-degree networks). But even the choice of a bounded-degree network does not ensure a modest use of space. All the networks in the references cited so far have a rather high degree of interconnection, so while the processor hardware may be kept in check, wiring may be overwhelming and space usage high in a system of very many processors. In VLSI technologies, particularly, space and interconnection (pin-out) are important.

A few examples illustrate some of the difficulties in finding universal networks when chip area is of concern. Consider first the use of a hypercube (or butterfly) as a universal network. An n-processor hypercube can simulate any bounded-degree network R of n processors by routing (partial) permutations. That is, since each processor of R can communicate with all of its neighbors in unit time, the hypercube can simulate R by sending at most a constant number of messages from each processor. Simulation in $O(\lg n)$ time (in the word model) follows from Valiant's scheme for routing on hypercubes [103]. But an n-processor hypercube requires $\Omega(n^2)$ layout area by application of Thompson's bisection-width lower bound on area [95, 96]. Thus, an area-A hypercube cannot simulate all area-A networks efficiently. For example, an area-A mesh cannot

[1] Some of these sources treat the subject indirectly by providing results about sorting networks.

be simulated in polylogarithmic time by an area-A hypercube, since the mesh has $\Theta(A)$ processors and the hypercube only $\Theta(\sqrt{A})$ processors. Unfortunately, meshes are not universal with respect to area either; they cannot simulate trees. An area-A tree has $\Theta(A)$ processors using the H-tree layout and diameter[2] $O(\lg A)$, but the mesh has diameter $\sqrt{A}$. Thus, some pair of processors in the tree will be too far apart in the mesh to permit a simulation in polylogarithmic time. Finally, we can consider trees simulating meshes, to see that trees also fail to be universal with respect to area. Any bisection of an area-A mesh must cut $\sqrt{A}$ wires, and simultaneous communication across these wires cannot be supported in polylogarithmic time by a tree with its $O(1)$ bisection.

Leiserson [62] pioneered a combination of the interest in general-purpose computation with the concern for physical space as a measure of hardware cost. He introduced a class of routing networks referred to as fat-trees and showed that an appropriate n-processor network from this class can simulate (off-line) any other routing network on the same number of processors and occupying the same volume (or area) with only $O(\lg^3 n)$ factor degradation in the time required.

This chapter lays the groundwork for further development of the notion of area-universality using variations on the fat-tree architecture. Section 2.1 introduces the fat-tree network. Section 2.2 discusses underlying modeling assumptions used in the study of universality and sketches the means by which fat-trees are proved universal in preparation for the detailed studies which follow.

2.1 Fat-Trees

The basic fat-tree network as introduced by Leiserson [62] and based on Leighton's tree-of-meshes graph [56] is illustrated in Figure 2-1. A set of n processors are located at the leaves of a complete binary tree. Each edge of the underlying tree corresponds to two *channels* of the fat-tree: one from parent to child, the other from child to parent. Unlike

[2]The diameter is the maximum over all processor pairs of the length (in number of edges) of the shortest path connecting the two processors.

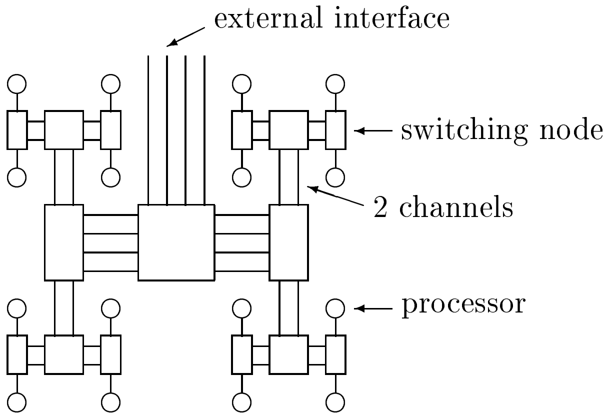

Figure 2-1: The organization of a fat-tree. Processors are located at the leaves, and the internal nodes contain concentrator switches. The channels between internal nodes consist of bundles of wires.

a normal tree which is "skinny all over," in a fat-tree, each channel consists of a bundle of wires. The number of wires in a channel c is called its capacity, denoted by $\mathrm{cap}(c)$. Each internal node of the fat-tree contains circuitry that switches messages from incoming to outgoing channels.

The channel capacities of a fat-tree determine the amount of hardware required to build it. The greater the capacities of the channels, the greater the communication potential, and also, the greater the hardware cost of an implementation of the network. The idea of fat-trees is to take advantage of a principle of locality in much the same way as does the telephone network: by using only slightly more hardware than that required to support fast *nonlocal* communication among a set of processors, much additional *local* communication among a larger set of processors can be supported.

Fat-trees can also be built using switches with a constant number of inputs and outputs [44] as in Figure 2-2. Since these switches are reminiscent of butterfly switches, we refer to such a fat-tree as a butterfly fat-tree. There is still an underlying channel structure corresponding to the channels shown in Figure 2-1. By using two types of constant-size switches, it is possible to build butterfly fat-trees with essentially arbitrary channel capacities as illustrated in Figure 2-3 borrowed from Leiserson [64].

The mechanics of routing on fat-trees will be further elaborated in Chapter 3; the remainder of this chapter sketches the means by which fat-trees are shown universal in

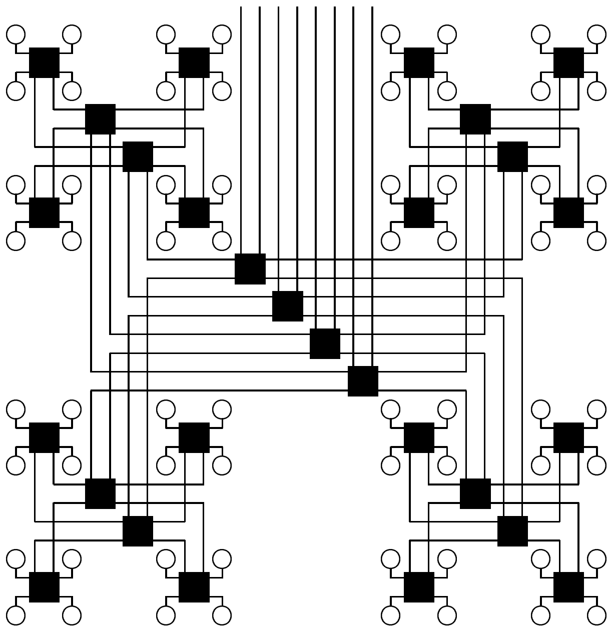

Figure 2-2: Another fat-tree design. The switches in this structure have constant size.

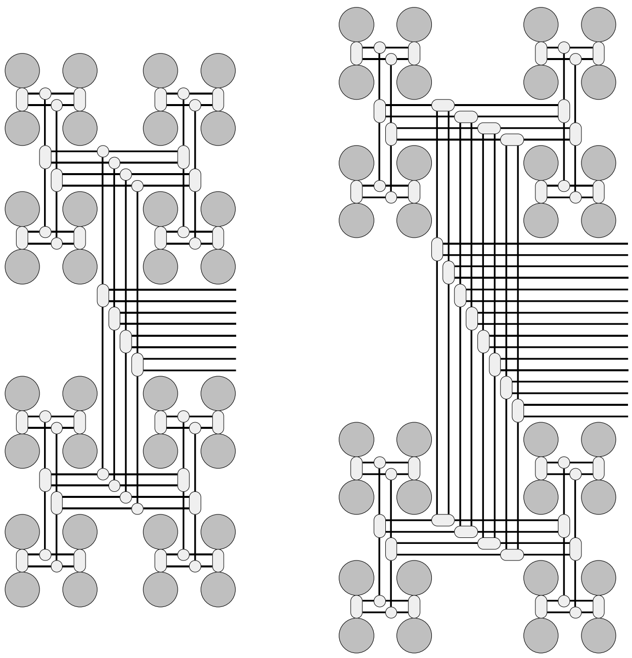

Figure 2-3: A scalable fat-tree. Essentially arbitrary channel capacities can be obtained by using two types of constant-size switches.

order to motivate the later analyses of routing time and hardware cost.

2.2 Universality

This section begins by discussing basic modeling assumptions about VLSI technologies and the operation of parallel computers. It also outlines the proof of universality for fat-trees which will be developed in detail in the succeeding chapters.

A few important but reasonable assumptions about time and space, stemming from the discussion of VLSI models in Section 1.1, merit reemphasis. First of all, the technology being used determines a minimum feature size, which is our unit of space. (Sometimes processor size will be taken as the measure of unit space for simplicity, but ultimately we will seek to explicitly account for processor size in terms of the more fundamental unit of space.) Second there is a fundamental lower bound on the time to switch a wire of unit length, which will be used as our basic unit of time. In VLSI technologies, such a bound is determined by the capacitance and resistance of a minimum-size transistor and the necessity of increasing capacitance if resistance is to be reduced. In fact, we will initially assume that this is the only source of delay and that transmission along any wire can be accomplished in unit time, but later we will relax this assumption. In any case, we assume that the number of bits which can leave an area of a chip in unit time is proportional to the perimeter of the area. (In three dimensions, the number of bits which can leave a region of space in unit time is proportional to the surface area of the region.)

The basic mode of operation assumed for parallel computers will be as in the distributed random-access machine (DRAM) model of Leiserson and Maggs [65]. All memory is local to the processors, with each processor holding a small number of $O(\lg n)$-bit registers. A processor can read, write, and perform arithmetic and logical functions on values stored in its local memory. It can also read and write memory in other processors by routing messages through an underlying network. (Other models are easily accommodated; for example, a "dance-hall" model which places processors and memory modules in different locations on a network can be viewed as a special case of the model considered here.)

It is also convenient to assume that operation of any competing network is divided into separate phases of intraprocessor computation and interprocessor communication. Thus, to bound asymptotic simulation time, it will suffice to take the maximum of the overheads for simulation of the computation and simulation of the communication. In fact, this approach is valid even if the competing network interleaves computation and communication in a more complicated fashion. The ability to simulate such "asynchronous" competition is more formally described and proved in Section 4.5.

All of the universality proofs to be presented have the following basic outline. Given a competing network, we begin by recursively bisecting it until each piece of the competing network contains at most one processor. Then we can bound the number of messages which can flow into or out of pieces of the network in unit time. We assume that the competing network is a square of area A; for most of the results it makes little difference if the competing network is not square.

Given a competing network of area A, the recursive decomposition is as in Figure 2-4, with cuts alternating between the two dimensions as we go from one level to the next in the decomposition. The perimeter of any piece at level l in the decomposition is $O(\sqrt{A}/2^{l/2})$. The recursive bisection gives rise to a decomposition tree as represented in Figure 2-5, where the leaves represent the pieces at the bottom level of the decomposition, and those which are numbered represent pieces containing a processor.

Chapter 4 shows that for comparison between a fat-tree and networks of essentially equal area, it is possible to construct a fat-tree with enough processors to directly map the leaves of the decomposition tree to fat-tree processors. In other cases, it may be necessary to obtain a balanced decomposition tree, that is, one in which each cut of the competing network splits the processors evenly between the two halves. Fortunately general results by Bhatt and Leighton [13], and by Leiserson [62] in a cleaner form for our purposes, show that any decomposition tree may be converted to a balanced decomposition tree

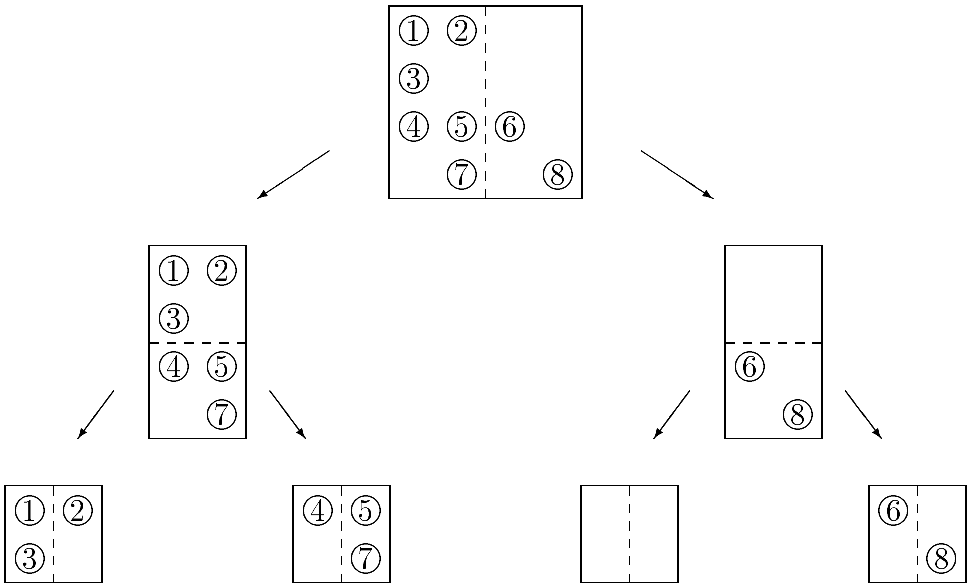

Figure 2-4: For analysis, a competing network is recursively bisected and a bound is established on the message traffic through the perimeter of pieces in the decomposition. The numbered circles represent processors in the competing network.

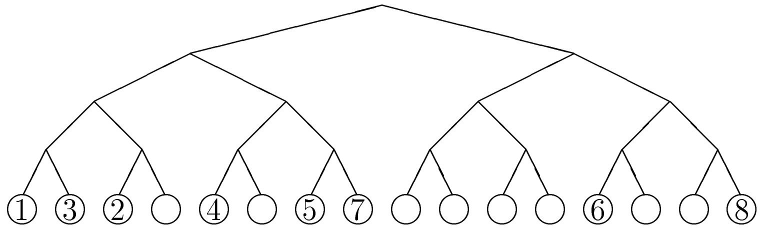

Figure 2-5: Decomposition tree of competing network. The recursive bisection proceeds until each piece of the competing network contains one processor (represented by a numbered leaf) or none.

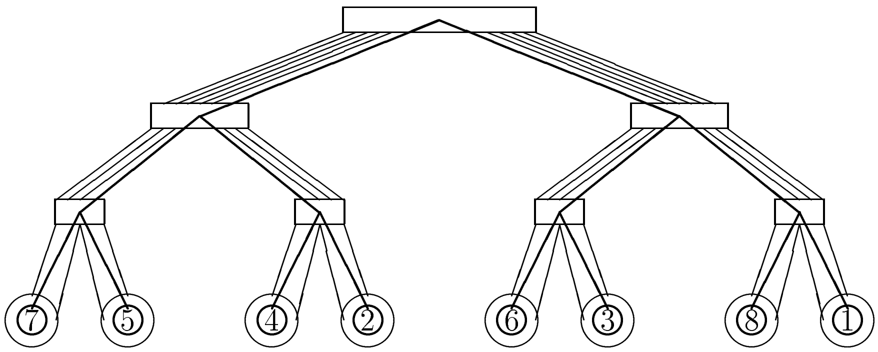

Figure 2-6: Mapping the decomposition tree of a competing network to a fat-tree.

with cuts of size differing by only a constant factor.

Once a suitable decomposition tree is obtained, competing processors may be mapped onto fat-tree processors in the natural fashion as shown in Figure 2-6. The channel capacities in the fat-tree are chosen to be sufficiently slowly growing as to keep the area of the fat-tree modest but sufficiently quickly growing as to roughly match the bound on the amount of message traffic crossing the perimeter of subpieces of the competing network. Chapter 4 discusses this in more detail and shows that a fat-tree can be constructed so that the messages delivered in unit time by a competing network of comparable area will place a load of at most $O(\lg n \operatorname{cap}(c))$ messages that need to pass through any channel c in the fat-tree.

The type of bound just given for the communications burden imposed on a fat-tree by a message set is referred to as the *load factor* of the message set, since the number of messages passing through any channel is bounded by some factor times the capacity. As long as we can show that a message set can be delivered in time which is moderate relative to the load factor, the proof of universality will be complete. It is this routing task which is covered in Chapter 3.

Chapter 3

Routing on Fat-Trees

This chapter describes algorithms for routing messages on fat-trees. The running times of these algorithms are expressed as a function of the *load factor* of a set of messages to be routed, the load factor being a lower bound on the routing time. The main contribution of this chapter is an algorithm (for routing on an n-processor fat-tree) that routes any set of messages with load factor λ in $O(\lambda + \lg n \lg\lg n)$ delivery cycles (routing attempts) with probability $1 - O(1/n)$. This chapter is drawn from a joint paper with Charles Leiserson of MIT [44].

An issue that any routing algorithm for a fat-tree must face is that some communication patterns among the processors are harder than others. For example, suppose the channel capacity between the two halves of a fat-tree is $\Theta(n^{1/2})$, where n is the number of processors, and suppose each processor sends a message to a processor in the other half. Since the number of messages that must pass through the root is n and the capacity is $\Theta(n^{1/2})$, the time required by the network to deliver all the messages is $\Omega(n^{1/2})$ because of congestion. In contrast, there are many communication patterns among n processors with less global communication that can be implemented in subpolynomial time. For example, each processor might simply send a message to the processor with the same parent in the tree. This pattern produces no congestion, and the entire communication pattern can be accomplished in the time required to send a single one of the messages.

The routing algorithms for fat-trees presented in this chapter are randomized algorithms which are analyzed in terms of a measure of congestion called the *load factor*. The load factor of a set of messages is the largest ratio over all channels in the fat-tree of the number of messages that must pass through the channel divided by the capacity of the channel. The load factor of a set of messages is thus a lower bound on the time to deliver the messages.

The algorithm discussed most extensively in this chapter can deliver a set of messages with load factor λ in $O(\lambda + \lg n \lg\lg n)$ delivery cycles (routing attempts) with high probability. This improves on Leiserson's bound of $O(\lambda \lg n)$ for the off-line situation where the set of messages is known in advance and the problem is to schedule their delivery [62]. Since delivery cycles will be shown to run in $O(\lg^2 n)$ time, the total running time of the algorithm is $O(\lambda \lg^2 n + \lg^3 n \lg\lg n)$.

Other routing algorithms, which address other issues, are also presented. In the important special case that all channel capacities in the fat-tree are of size $\Omega(\lg n)$, the messages can be delivered in $O(\lambda)$ delivery cycles, for a total running time of $O(\lambda \lg^2 n)$. In the case of the butterfly fat-tree, an algorithm is presented which uses $O(\lambda \lg^2 n)$ delivery cycles, but each delivery cycle requires only $O(\lg n)$ time so that the total routing time is $O(\lambda \lg^3 n)$.

The analysis of these randomized routing algorithms makes no assumptions about the statistical distribution of messages, except insofar as it affects the load factor. Moreover, the algorithms are not restricted to permutation routing or situations where each processor can only send or receive a constant number of messages, as is common in the literature. We consider the general situation where each processor can send and receive polynomially many messages.

These routing algorithms also differs from others in the literature in the way randomization is used. Unlike the algorithms of Valiant [103], Valiant and Brebner [104], Aleliunas [5], Upfal [99] and Pippenger [80], for example, they involve little or no randomization with respect to paths taken by messages. For example, Valiant's classic scheme for routing on a hypercube sends each message to a randomly chosen intermediate destination and, from there, to its true destination. On a fat-tree, such a technique would likely convert communication patterns with good locality into ones with much global communication. Instead of being based primarily on the choice of random paths for messages to traverse, the algorithms presented here repeatedly attempt to deliver a randomly chosen subset of the messages. A by-product of this strategy is that the algorithms require no intermediate buffering of messages.

All of these algorithms are based on the type of communications scheme for fat-trees originally used by Leiserson [62], which, as explained below, requires only switches of a very simple nature. We consider communication through the fat-tree network to be synchronous, bit serial, and batched. By synchronous, we mean that the system is globally clocked. By bit serial, we mean that the messages can be thought of as bit streams. Each message snakes its way through the wires and switches of the fat-tree, with leading bits of the message setting switches and establishing a path for the remainder to follow. By batched, we mean the messages are grouped into *delivery cycles*. During a delivery cycle, the processors send messages through the network. Each message attempts to establish a path from its source to its destination. Since some messages may be unable to establish connections during a delivery cycle, each successfully delivered message is acknowledged through its communication path at the end of the cycle. Rather than buffering undelivered messages, we simply allow them to try again in a subsequent delivery cycle. The routing algorithm is responsible for grouping the messages into delivery cycles so that all the messages are delivered in as few cycles as possible.

The mechanics of routing messages in a fat-tree are similar to routing in an ordinary tree. For each message, there is a unique path from its source processor to its destination processor in the underlying complete binary tree, which can be specified by a relative address consisting of at most $2\lg n$ bits telling whether the message turns left or right at each internal node. Within each node of the fat-tree, the messages destined for a given output channel are *concentrated* onto the available wires of that channel. This concentration may result in "lost" messages if the number of messages destined for the output channel exceeds the capacity of the channel. We assume, however, that the concentrators within the node are ideal in the sense that no messages are lost if the number of messages destined for a channel is less than or equal to the capacity of the channel. (No stronger properties, for example any sort of randomization, are required of the switches.) Such a concentrator can be built, for example, with a logarithmic-depth sorting network [3, 4]. A somewhat more practical logarithmic-depth circuit can be built by combining a parallel prefix circuit [53] with a butterfly (i.e., FFT, Omega) network. Detailed designs of concentrator switches using a different approach have been provided by Cormen and Leiserson [30]. With switches of logarithmic depth, the time to run each delivery cycle is $O(\lg^2 n)$ bit times, making the natural assumption that messages are $O(\lg n)$ bits long.[1] (In the butterfly fat-tree, the switches are of constant depth, and the time to run a delivery cycle is $O(\lg n)$ bit times.)

The performance of any routing algorithm for a fat-tree depends on the locality of communication inherent in a set of messages. The locality of communication for a message set M can be summarized by a measure $\lambda(M)$ called the *load factor*, which we define in a more general network setting.

> **Definition:** Let R be a routing network. A set S of wires in R is a (directed) *cut* if it partitions the network into two sets of processors A and B such that every path from a processor in A to a processor in B contains a wire in S. The *capacity* $\text{cap}(S)$ is the number of wires in the cut. For a set of messages M, define the *load* $\text{load}(M,S)$ of M on a cut S to be the number of messages in M from a processor in A to a processor in B. The *load factor* of M on S is
>
> $$\lambda(M,S) = \frac{\text{load}(M,S)}{\text{cap}(S)} ,$$

[1] In this thesis, times are generally measured in terms of bit operations, rather than word operations, to better reflect actual costs.

and the *load factor* of M on the entire network R is

$$\lambda(M) = \max_S \lambda(M,S) \ .$$

The load factor of a set of messages on a given network provides a lower bound on the time required to deliver all messages in the set.

For fat-trees, only cuts corresponding to channels need be considered to determine the load factor, as is shown by the following lemma.

Lemma 1 *The load factor of a set M of messages on a fat-tree is*

$$\lambda(M) = \max_c \lambda(M,c) \ ,$$

where c ranges over all channels of the fat-tree.

Proof. Any cut S must entirely contain at least one channel. Let us partition the wires in S into $S = c_1 \cup \cdots \cup c_l \cup w$, where $c_1, \ldots, c_l$ are the complete channels in S and w is the set of remaining wires in S. For convenience, let $x_i = \mathrm{load}(M, c_i)$ and $y_i = \mathrm{cap}(c_i)$. Assume without loss of generality that $\lambda(M, c_1) \geq \lambda(M, c_i)$ for $i = 1, \ldots, l$, which implies $x_1 y_i - x_i y_1 \geq 0$ for all i. The load factor of M on S is therefore

$$\begin{aligned} \lambda(M,S) &= \frac{x_1 + \cdots + x_l}{y_1 + \cdots + y_l + |w|} \\ &= \frac{x_1}{y_1} - \frac{(x_1 y_2 - x_2 y_1) + \cdots + (x_1 y_l - x_l y_1) + (x_1 \, |w|)}{y_1(y_1 + \cdots + y_l + |w|)} \\ &\leq \frac{x_1}{y_1} \\ &= \lambda(M, c_1) \end{aligned} \tag{3.1}$$

since each term in the numerator of the second term of (3.1) is nonnegative. ■

The remainder of this chapter presents and analyzes routing algorithms for the fat-tree and butterfly fat-tree. The general routing algorithm for fat-trees is given in Section 3.1 and its analysis in Section 3.2. Section 3.3 gives an existential lower bound for a class of naive greedy routing algorithms which shows that the greedy strategy is inferior to the randomized algorithm of Section 3.1 for worst-case inputs. Section 3.4 gives a routing algorithm for the special case in which all channels are of capacity $\Omega(\lg n)$ and proves its validity using preliminary parts of the analysis in Section 3.2. Finally, Section 3.5 provides an algorithm for routing on the butterfly fat-tree. It also discusses briefly two algorithms by other researchers [61, 76] for routing on butterfly fat-trees; these algorithms obtain better running times while using more complicated switches.

3.1 The General Routing Algorithm

This section gives the general randomized algorithm for routing a set M of messages on a fat-tree. The algorithm *RANDOM*, which is based on routing random subsets of the messages in M, is shown in Figure 3-1. It uses the subroutine *TRY-GUESS* shown in Figure 3-2. Section 3.2 provides a proof that on an n-processor fat-tree, the probability is at least $1-O(1/n)$ that *RANDOM* delivers all messages in M within $O(\lambda(M)+\lg n \lg\lg n)$ delivery cycles, if the two constants k_1 and k_2 appearing in the algorithm are properly chosen.

The basic idea of *RANDOM* is to pick a random subset of messages to send in each delivery cycle by independently choosing each message with some probability p. This type of message set merits a formal definition.

> **Definition:** A *p-subset* of M is a subset of M formed by independently choosing each message of M with probability p.

We will show in Section 3.2 that if p is sufficiently small, many of the messages in a p-subset are delivered because they encounter no congestion during routing. On the other hand, if p is too small, few messages are sent. *RANDOM* varies the probability p from cycle to cycle, seeking random subsets of M that contain a substantial portion of the messages in M, but that do not cause congestion.

The algorithm *RANDOM* varies the probability p because the load factor $\lambda(M)$ is

Algorithm *RANDOM*
send M
$U \leftarrow M - \{\text{messages delivered}\}$
$\lambda_{guess} \leftarrow 2$
while $k_1\lambda_{guess} < k_2 \lg n$ and $U \neq \emptyset$ **do**
    TRY-GUESS(λ_{guess})
    $\lambda_{guess} \leftarrow \lambda_{guess}^2$
endwhile
$\lambda_{guess} \leftarrow (k_2/k_1) \lg n \lg\lg n$
while $U \neq \emptyset$ **do**
    TRY-GUESS(λ_{guess})
    $\lambda_{guess} \leftarrow 2\lambda_{guess}$
endwhile

Figure 3-1: The randomized algorithm *RANDOM* for delivering a message set M on a fat-tree with n processors. This algorithm achieves the running times in Figure 3-3 with high probability if the constants k_1 and k_2 are appropriately chosen. Since the load factor $\lambda(M)$ is not known in advance, *RANDOM* makes guesses, each one being tried out by the subroutine *TRY-GUESS*.

procedure *TRY-GUESS*(λ_{guess})
$\lambda \leftarrow \lambda_{guess}$
while $\lambda > 1$ **do**
    for $i \leftarrow 1$ **to** $\max\{k_1\lambda, k_2 \lg n\}$ **do**
        independently send each message of U with probability $1/r\lambda$
        $U \leftarrow U - \{\text{messages delivered}\}$
    endfor
    $\lambda \leftarrow \lambda/2$
endwhile
send U
$U \leftarrow U - \{\text{messages delivered}\}$

Figure 3-2: The subroutine *TRY-GUESS* used by the algorithm *RANDOM* which tries to deliver the set U of currently undelivered messages. When $\lambda_{guess} \geq \lambda(U)$, this attempt will be successful with high probability, if the constants k_1 and k_2 are appropriately chosen. (The value r is the congestion parameter of the fat-tree defined in Section 3.2, which is typically a small constant.) In that case, λ is always an upper bound on $\lambda(U)$, which is at least halved in each iteration of the **while** loop. When the loop is finished, $\lambda(U) \leq 1$, so all the remaining messages can be sent.

not known. The overall structure of *RANDOM* is to guess the load factor and call the subroutine *TRY-GUESS* for each one. The subroutine *TRY-GUESS* determines the probability p based on *RANDOM*'s guess λ_{guess} and a parameter r, called the *congestion parameter* of the fat-tree, which is independent of the message set and which will be defined in Section 3.2. If λ_{guess} is an upper bound on the true load factor $\lambda(M)$, then with high probability, each iteration of the **while** loop in *TRY-GUESS* halves the upper bound on the load factor $\lambda(U)$ of the set U of undelivered messages, as will be shown in Section 3.2. When the loop is finished, we have $\lambda(U) \leq 1$, and all the remaining messages can be delivered in one cycle. The number of delivery cycles performed by *TRY-GUESS* is $O(\lg \lambda_{guess} \lg n)$ if $2 \leq \lambda_{guess} \leq \Theta(\lg n)$, and the number of cycles is $O(\lambda_{guess} + \lg n \lg \lg n)$ if $\lambda_{guess} = \Omega(\lg n)$.

RANDOM must make judicious guesses for the load factor because *TRY-GUESS* may not be effective if the guess is smaller than the true load factor. Conversely, if the guess is too large, too many delivery cycles will be performed. Since the amount of work done by *TRY-GUESS* grows as $\lg \lambda_{guess}$ when λ_{guess} is small, and as λ_{guess} when λ_{guess} is large, there are two main phases to *RANDOM*'s guessing. (These phases follow the handling of very small load factors, i. e., $\lambda(M) \leq 2$.)

In the first phase, the guesses are squared from one trial to the next. Once λ_{guess} is sufficiently large, we move into the second phase, and the guesses are doubled from one trial to the next. In each phase, the number of delivery cycles run by *TRY-GUESS* from one call to the next forms a geometric series. Thus, the work done in any call to *TRY-GUESS* is only a constant factor times all the work done prior to the call. With this guessing strategy, we can deliver a message set using only a constant factor more delivery cycles than would be required if we knew the load factor in advance.

3.2 Analysis of the Routing Algorithm

This section contains the analysis of *RANDOM*, the routing algorithm for fat-trees presented in Section 3.1. We shall show that the probability is $1 - O(1/n)$ that *RANDOM*

load factor	delivery cycles
$0 \le \lambda(M) \le 1$	1
$1 \le \lambda(M) \le 2$	$O(\lg n)$
$2 \le \lambda(M) \le \lg n \lg\lg n$	$O(\lg n \lg(\lambda(M)))$
$\lg n \lg\lg n \le \lambda(M) \le n^{O(1)}$	$O(\lambda(M))$

Figure 3-3: The number of delivery cycles required to deliver a message set M on a fat-tree with n processors. All bounds are achieved with probability $1 - O(1/n)$. The bounds on the number of delivery cycles can be summarized as $O(\lambda(M) + \lg n \lg\lg n)$.

delivers a set M of messages on an n-processor fat-tree in $O(\lambda(M) + \lg n \lg\lg n)$ delivery cycles (subject to mild conditions on the channel capacities). Figure 3-3 gives the tighter bounds that we actually prove.

We begin by stating two technical lemmas concerning basic probability. One is a combinatorial bound on the tail of the binomial distribution of the kind attributed to Chernoff [27], and the other is a general, but weak, bound on the probability that a random variable takes on values smaller than the expectation.

The first lemma is the Chernoff bound. Consider t independent Bernoulli trials, each with probability p of success. It is well known [37] that the probability that there are at least s successes out of the t trials is

$$B(s,t,p) = \sum_{k=s}^{t} \binom{t}{k} p^k (1-p)^{t-k} .$$

The lemma bounds the probability that the number of successes is larger than the expectation pt.

Lemma 2

$$B(s,t,p) \le \left(\frac{ept}{s}\right)^s .$$

Proof. The lemma follows from [103, p. 354]. ■

The second technical lemma bounds the probability that a bounded random variable takes on values smaller than the expectation.

Lemma 3 *Let $X \le b$ be a random variable with expectation μ. Then for any $w < \mu$, we have*

$$\Pr\{X \le w\} \le 1 - \frac{\mu - w}{b - w} .$$

Proof. The definition of expectation gives us

$$\mu \le w \Pr\{X \le w\} + b(1 - \Pr\{X \le w\}) ,$$

from which the lemma follows. ■

We now analyze the routing of a p-subset M' of a set M of messages. If the number $\mathrm{load}(M', c)$ of messages in M' that must pass through c is no more than the capacity $\mathrm{cap}(c)$, then no messages are lost by concentrating the messages into c. We shall say that c is *congested by* M' if $\mathrm{load}(M', c) > \mathrm{cap}(c)$. The next lemma shows that the likelihood of channel congestion decreases exponentially with channel capacity if the probability of choosing a given message in M is sufficiently small.

Lemma 4 *Let M be a set of messages on a fat-tree, let $\lambda(M)$ be the load factor on the fat-tree due to M, let M' be a p-subset of messages from M, and let c be a channel through which a given message $m \in M'$ must pass. Then the probability is at most $(ep\lambda(M))^{\mathrm{cap}(c)}$ that channel c is congested by M'.*

Proof. Channel c is congested by M' if $\mathrm{load}(M', c) > \mathrm{cap}(c)$. There is already one message from the set M' going through channel c, so we must determine a bound on the probability that at least $\mathrm{cap}(c)$ other messages go through c. Using Lemma 2 with $s = \mathrm{cap}(c)$ and $t = \mathrm{load}(M, c)$, the probability that the number of messages sent through channel c is greater than the capacity $\mathrm{cap}(c)$ is less than

$$\begin{aligned} B(\mathrm{cap}(c), \mathrm{load}(M, c), p) &\le \left(\frac{ep\,\mathrm{load}(M, c)}{\mathrm{cap}(c)}\right)^{\mathrm{cap}(c)} \\ &\le (ep\lambda(M))^{\mathrm{cap}(c)} . \end{aligned}$$

■

The next lemma will analyze the probability that a given message of a p-subset of M gets delivered. In order to do the analysis, however, we must select p small enough so that it is likely that the message passes exclusively through uncongested channels. The choice of p depends on the capacities of channels in the fat-tree. For convenience, we define a parameter of the capacities that will enable us to choose a suitable upper bound for p.

Definition: The *congestion parameter* of a fat-tree is the smallest positive value r such that for each simple path $c_1, c_2, \ldots, c_l$ of channels in the fat-tree, we have

$$\sum_{k=1}^{l} \left(\frac{e}{r}\right)^{\mathrm{cap}(c_k)} \leq \frac{1}{2} .$$

The congestion parameter is generally quite small. For any fat-tree based on a complete binary tree, the longest simple path is at most $2\lg n$, where n is the number of processors, and thus we have $r \leq 4e\lg n$. For universal fat-trees (discussed in Chapter 4), the congestion parameter is a constant because the capacities of channels grow exponentially as we go up the tree. (All we really need is arithmetic growth in the channel capacities.) The congestion parameter is also constant for any fat-tree based on a complete binary tree if all the channels have capacity $\Omega(\lg\lg n)$. Our analysis of *RANDOM* treats the congestion parameter r as a constant, but the analysis does not change substantially for other cases.

We now present the lemma that analyzes the probability that a given message gets delivered.

Lemma 5 *Let M be a set of messages on a fat-tree that has congestion parameter r, let $\lambda(M)$ be the load factor on the fat-tree due to M, and let m be an arbitrary message in M. Suppose M' is a p-subset of M, where $p \leq 1/r\lambda(M)$. Then if M' is sent, the probability that m gets delivered is at least $\frac{1}{2}p$.*

Proof. The probability that $m \in M$ is delivered is at least the probability that $m \in M'$ times the probability that m passes exclusively through uncongested channels. The probability that $m \in M'$ is p, and thus we need only show that, given $m \in M'$, the probability is at least $\frac{1}{2}$ that every channel through which m must pass is uncongested. Let c_1, c_2, $\ldots$, c_l be the channels in the fat-tree through which m must pass. The probability that channel c_k is congested is less than $(e/r)^{\mathrm{cap}(c_k)}$ by Lemma 4. The probability that at least one of the channels is congested is, therefore, less than

$$\sum_{k=1}^{l} \left(\frac{e}{r}\right)^{\mathrm{cap}(c_k)} \leq \frac{1}{2} ,$$

by definition of the congestion parameter. Thus, the probability that none of the channels are congested is at least $\frac{1}{2}$. ∎

We now focus our attention on *RANDOM* itself. The next lemma analyzes the innermost loop (lines 3–6) of *RANDOM*'s subroutine *TRY-GUESS*. At this point in the algorithm, there is a set U of undelivered messages and a value for λ. The lemma shows that if λ is indeed an upper bound on the load factor $\lambda(U)$ of the undelivered messages when the loop begins, then $\lambda/2$ is an upper bound after the loop terminates. This lemma is the crucial step in showing that *RANDOM* works.

> **Lemma 6** *Let U be a set of messages on an n-processor fat-tree with congestion parameter r, and assume $\lambda(U) \leq \lambda$. Then after lines 3–6 of RANDOM's subroutine TRY-GUESS, the probability is at most $O(1/n^2)$ that $\lambda(U) > \frac{1}{2}\lambda$.*

Proof. The idea is to show that the load factor of an arbitrary channel c remains larger than $\frac{1}{2}\lambda$ with probability $O(1/n^3)$. Since the channel c is chosen arbitrarily out of the $4n-2$ channels in the fat-tree, the probability is at most $O(1/n^2)$ that any of the channels is left with a load factor larger than $\frac{1}{2}\lambda$.

For convenience, let C be the subset of messages that must pass through channel c and are undelivered at the beginning of the innermost loop in *RANDOM*. Let $C_0 = C$, and for $i \geq 1$, let $C_i \subseteq C_{i-1}$ denote the set of undelivered messages at the end of the

ith iteration of the loop. Notice that we have $\lambda(C_i, c) = |C_i| / \operatorname{cap}(c)$, since we have $|C_i| = \operatorname{load}(C_i, c)$ by definition.

We now show there exist values for the constants k_1 and k_2 in line 3 of *TRY-GUESS* such that for $z = \max\{k_1\lambda, k_2 \lg n\}$, the probability is $O(1/n^3)$ that $\lambda(C_z, c) > \frac{1}{2}\lambda$, or equivalently, that

$$|C_z| > \frac{1}{2}\lambda \operatorname{cap}(c) . \tag{3.2}$$

It suffices to prove that the probability is $O(1/n^3)$ that fewer than $\frac{1}{2}|C|$ messages from C are delivered during the z cycles under the assumption that $|C_i| > \frac{1}{2}\lambda \operatorname{cap}(c)$ for $i = 0, 1, \ldots, z-1$. The intuition behind the assumption $|C_i| > \frac{1}{2}\lambda \operatorname{cap}(c)$ is that otherwise, the load factor on channel c is already at most $\frac{1}{2}\lambda$ at this step of the iteration. The reason we need only bound the probability that fewer than $\frac{1}{2}|C|$ messages are delivered during the z cycles is that inequality (3.2) implies that the number of messages delivered is fewer than $|C| - \frac{1}{2}\lambda \operatorname{cap}(c) \le |C| - \frac{1}{2}\lambda(C, c) \operatorname{cap}(c) \le \frac{1}{2}|C|$.

We shall establish the $O(1/n^3)$ bound on the probability that at most $\frac{1}{2}|C|$ messages are delivered in two steps. For convenience, we shall call a cycle *good* if at least $\operatorname{cap}(c)/8r$ messages are delivered, and *bad* otherwise. In the first step, we bound the probability that a given cycle is bad. The expected number of messages delivered in any given cycle is the product of the number of messages that remain to be delivered and the probability that any of these messages is successfully delivered. Using Lemma 5 with $p = 1/r\lambda \le 1/r\lambda(U) \le 1/r\lambda(C_i)$ in conjunction with the assumption that $|C_i| > \frac{1}{2}\lambda \operatorname{cap}(c)$, we can conclude that the expected number of messages delivered in any given cycle is greater than $\frac{1}{2r\lambda}\frac{1}{2}\lambda \operatorname{cap}(c) = \operatorname{cap}(c)/4r$. Then by Lemma 3, the probability that a given cycle is bad is at most $1 - 1/(8r-1) < 1 - 1/8r$.[2]

The second step bounds the probability that a substantial fraction of the z delivery cycles are bad. Specifically, we show that the probability is $1 - O(1/n^3)$ that at least

[2]We use the weak bound of Lemma 3 because we cannot assume that the probability that a message is delivered in a given cycle is independent of the probabilities for other messages. In practice, one would anticipate that the dependencies between messages are weak, and that the algorithm would be effective with much smaller values for the constants k_1 and k_2 than we prove here.

some small constant fraction q of the z cycles are good. By picking $k_1 = 4r/q$, which implies $z \geq 4r\lambda/q$, at least $qz\,\mathrm{cap}(c)/8r \geq \frac{1}{2}|C|$ messages are delivered.

We bound the probability that at least $(1-q)z$ of the z cycles are bad by using a counting argument. There are $\binom{z}{(1-q)z}$ ways of picking the bad cycles, and the probability that a cycle is bad is at most $1-1/8r$. Thus, the probability that at most $\frac{1}{2}|C|$ messages are delivered is

$$\begin{aligned}
\Pr\left\{\leq \tfrac{1}{2}|C| \text{ messages delivered}\right\} &\leq \binom{z}{(1-q)z}\left(1-\frac{1}{8r}\right)^{(1-q)z} \\
&\leq \left(q^q(1-q)^{1-q}\right)^{-z}\left(1-\frac{1}{8r}\right)^{(1-q)z} \qquad (3.3)\\
&\leq 2^{-z/12r}\,, \qquad (3.4)
\end{aligned}$$

where (3.3) follows from Stirling's approximation (for sufficiently large z), and (3.4) follows from choosing $q = 1/100r\ln r$ and performing algebraic manipulations. Since $z = \max\{k_1\lambda, k_2\lg n\}$, if we choose $k_2 = 36r$, the probability that fewer than $\frac{1}{2}|C|$ messages are delivered is at most $1/n^3$. ■

Now we can analyze *RANDOM* as a whole.

Theorem 7 *For any message set M on an n-processor fat-tree, the probability is at least $1 - O(1/n)$ that RANDOM delivers all the messages of M within the number of delivery cycles specified by Figure 3-3.*

Proof. First, we will show that if $\lambda_{guess} \geq \lambda(M)$, the probability is at most $O(1/n)$ that the loop in lines 2 through 8 of *TRY-GUESS* fails to yield $\lambda(U) \leq 1$. Initially, $\lambda \geq \lambda(U)$, and we know from Lemma 6 that the probability is at most $O(1/n^2)$ that any given iteration of the loop fails to restore this condition as λ is halved. Since there are $\lg\lambda_{guess}$ iterations of the loop, we need only make the reasonable assumption that λ_{guess} is polynomial in n to obtain a probability of at most $O(1/n)$ that $\lambda(U)$ remains greater than 1 after all the iterations of the loop. That this assumption holds can be verified for each of the cases below by noting that $\lambda(M)$ is at most polynomial in n and that λ_{guess} is never much larger than $\lambda(M)$.

Now we just need to count the number of delivery cycles that have been completed by the time we call *TRY-GUESS* with a λ_{guess} such that $\lambda(M) \leq \lambda_{guess}$. Let us denote by λ^*_{guess} the first λ_{guess} that satisfies this condition, and then break the analysis down into cases according to the value of $\lambda(M)$.

For $\lambda(M) \leq 1$, we do not actually even call *TRY-GUESS*. We need only count the one delivery cycle executed in line 1 of *RANDOM*.

For $1 < \lambda(M) \leq 2$, we need add only the $k_2 \lg n$ cycles executed when we call *TRY-GUESS*(2).

For $2 < \lambda(M) < (k_2/k_1)\lg n$, the number of delivery cycles involved in each execution of *TRY-GUESS* is $O(\lg \lambda_{guess} k_2 \lg n)$, since we perform $O(\lg \lambda_{guess})$ iterations of the loop in lines 2–8 of *TRY-GUESS*, each containing $k_2 \lg n$ iterations of the loop in lines 3–6. The value of λ^*_{guess} is at most $(\lambda(M))^2$, so the number of delivery cycles is $O(\lg n \lg(\lambda(M))^2)$ for the last guess, $O(\lg n \lg \lambda(M))$ for the second-to-last guess, $O(\lg n \lg \sqrt{\lambda(M)})$ for the third-to-last guess, and so on. The total number of delivery cycles is, therefore,

$$\begin{aligned}\sum_{0 \leq i \leq 1+\lg\lg\lambda(M)} O(\lg n \lg(\lambda(M))^{2^{1-i}}) &= \sum_{0 \leq i \leq 1+\lg\lg\lambda(M)} O(2^{1-i} \lg n \lg(\lambda(M))) \\ &= O(\lg n \lg \lambda(M)),\end{aligned}$$

since the series is geometric.

For $\lambda(M) > (k_2/k_1)\lg n$, the number of delivery cycles executed by the time we reach line 8 of *RANDOM* is $O(\lg n \lg\lg n)$ according to the preceding analysis, and then we must continue in the quest to reach λ^*_{guess}. If $\lambda(M) \leq (k_2/k_1)\lg n \lg\lg n$, then we need only add the $O(\lg n \lg\lg n) = O(\lg n \lg \lambda(M))$ delivery cycles involved in the single call *TRY-GUESS*$((k_2/k_1)\lg n \lg\lg n)$.

If $\lambda(M) > (k_2/k_1)\lg n \lg\lg n$, the number of delivery cycles executed before reaching line 8 is $O(\lg n \lg\lg n)$ as before, which is $O(\lambda(M))$. We must then add $O(\lambda_{guess})$ cycles for each call of *TRY-GUESS* in line 10. Since λ^*_{guess} is at most $2\lambda(M)$, the total additional number of delivery cycles is

$$\sum_{0\le i\le t} O(2^{1-i}\lambda(M)) = O(\lambda(M)),$$

where $t = 1 + \lg(k_1\lambda(M)/k_2 \lg n \lg\lg n)$. The total number of delivery cycles is thus $O(\lambda(M))$. ■

The $1 - O(1/n)$ bound on the probability that *RANDOM* delivers all the messages can be improved to $1 - O(1/n^k)$ for any constant k by choosing $k_2 = 12(k+2)r$, or by simply running the algorithm through more choices of λ_{guess}.

We can also use *RANDOM* to obtain a routing algorithm that guarantees to deliver all the messages in finite time, and whose expected number of delivery cycles is as given in Figure 3-3. We simply interleave *RANDOM* with any routing strategy that guarantees to deliver at least one message in each delivery cycle. If the number of messages is bounded by some polynomial n^k, then we choose k_2 such that *RANDOM* works with probability $1 - O(1/n^k)$.

3.3 Theoretical Deficiency of Greedy Strategies

It is natural to wonder whether a simple greedy strategy of sending all undelivered messages on each delivery cycle, and letting them battle their ways through the switches, might be as effective as *RANDOM*, which we have shown to work well on every message set. As a practical matter, a greedy strategy may be a good choice, but it seems difficult to obtain tight bounds on the running time of greedy strategies. In fact, we show in this section that no naive greedy strategy works as well as *RANDOM* in terms of asymptotic running times. For simplicity, we restrict our proof to deterministic strategies and comment later on the extension to probabilistic ones. Specifically, we show that for a wide class of deterministic greedy strategies, there exist n-processor fat-trees and message sets with load factor λ such that $\Omega(\lambda \lg n)$ delivery cycles are required. Thus, if λ is asymptotically larger than $\lg\lg n$, the greedy strategy is worse than *RANDOM*, which essentially

while $M \neq \emptyset$ **do**
    send M
    $M \leftarrow M - \{\text{messages delivered}\}$
endwhile

Figure 3-4: The algorithm *GREEDY* for delivering a message set M. This algorithm repeatedly sends all undelivered messages. The performance is highly dependent on the behavior of the switches.

guarantees $O(\lambda + \lg n \lg\lg n)$ delivery cycles for any set of messages. The lower-bound proof for greedy routing is based on an idea due to F. M. Maley [70].

Figure 3-4 shows the greedy algorithm. The code for *GREEDY* does not completely specify the behavior of message routing on a fat-tree because the switches have a choice as to which messages to drop when there is congestion. (The processors also have this choice, but we shall think of them as being switches as well.) In the analysis of *RANDOM*, we presumed that all messages in a channel are lost if the channel is congested. To completely specify the behavior of *GREEDY*, we must define the behavior of switches when channels are congested.

The lower bound for *GREEDY* covers a wide range of switch behaviors. Specifically, we assume the switches have two properties:

1. Each switch is greedy in that it only drops messages if a channel is congested, and then only the minimum number necessary.

2. Each switch is *oblivious* in that decisions on which messages to drop are not based on any knowledge of the message set other than the presence or absence of messages on the switch's input lines.

We define the switches of a fat-tree to be *admissible* if they have these two properties. The conditions are satisfied, for example, by switches that drop excess messages at random, or by switches that favor one input channel over another. An admissible switch can even base its decisions on previous decisions, but it cannot predict the future or make decisions based on knowing what (or how many) messages it or other switches have dropped. (The definition of oblivious in property 2 can be weakened to include an even wider range of switch behaviors without substantially affecting our results.)

At this point, we restrict attention to deterministic greedy strategies and present the lower-bound theorem for *GREEDY* operating on an area-universal fat-tree. The theorem can be extended to a variety of other fat-trees. A discussion of the extension to probabilistic greedy strategies follows the proof of the theorem.

> **Theorem 8** *Consider an n-processor area-universal fat-tree with deterministic admissible switches whose channel capacities are* 1 *nearest the processors and double at every other level going up the tree. Then there exist message sets with load factor λ for which GREEDY requires $\Omega(\lambda \lg n)$ delivery cycles.*

Proof. For any $\lambda \geq 12$, we will construct a "bad" message set M_n on the n processors of the fat-tree by induction on the subtrees. The message set M_n will consist entirely of messages to be routed out of the root and will satisfy the following three properties:

1. The message set M_n has load factor at most λ.
2. The root channel of the fat-tree is full for the first $\frac{1}{3}\lambda$ delivery cycles.
3. A total of at least $\frac{1}{6}\lambda + \frac{1}{12}\lambda \lg n$ delivery cycles[3] are required to deliver all the messages in M_n.

For the base case we consider a subtree with 1 processor, that is, a leaf connected to a channel of capacity 1. The bad message set M_1 consists of λ messages to be sent from the single processor. The properties are satisfied since the root channel is congested throughout the first $\frac{1}{3}\lambda$ delivery cycles, and at least $\frac{1}{6}\lambda$ delivery cycles are needed to deliver all the messages.

We next show that we can construct the bad message set M_n assuming that we can construct a bad message set for a subtree of $n/4$ processors. The construction uses an adversary argument. First, we specify the pattern of inputs seen by the root switch of the

[3] Without loss of generality, we assume henceforth that $\frac{1}{12}\lambda$ is integral, since we could otherwise use $\lfloor \frac{1}{12}\lambda \rfloor$ with only a constant factor change.

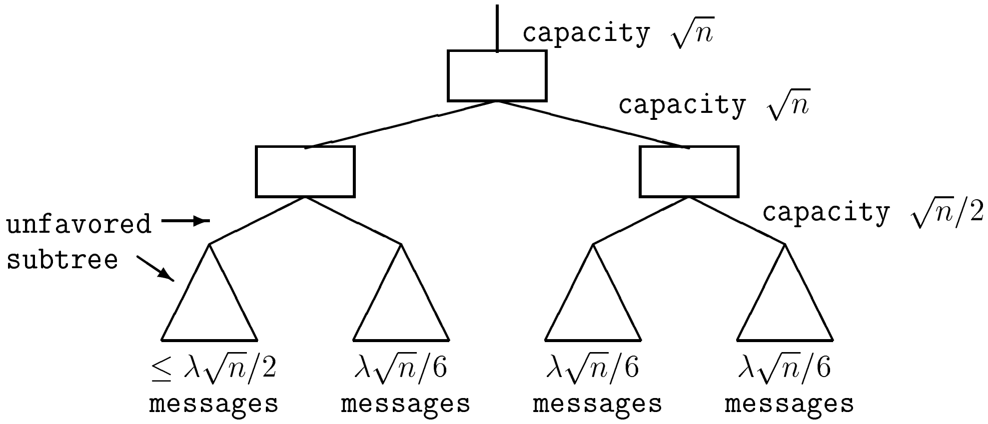

Figure 3-5: Construction of M_n for the proof of Theorem 8. The subtree from which the fewest number of messages have been delivered by a certain time is loaded with the largest number of messages.

fat-tree during certain delivery cycles, without giving any indication of how that input pattern can be achieved. Then since we have given enough information to determine the behavior of the root switch during these cycles, the root switch must announce which messages it passes through to its output. Finally, we give a construction for a message set that achieves the input pattern we called for in the first step. We take advantage of the announced behavior of the root switch in order to ensure that the message set also satisfies properties 1, 2, and 3.

We begin by calling for the input channels of the root switch of the fat-tree to be full for t delivery cycles, where t is $\frac{1}{3}\lambda$. If this is achieved, the total number of messages removed from the fat-tree during the first t delivery cycles is $m = \frac{1}{3}\lambda\sqrt{n}$, since the root capacity is $\sqrt{n}$ and the root switch is greedy. Also, as mentioned before, the specified input pattern determines the behavior of the root switch because the switch is oblivious.

The behavior of the root switch determines how many of the m messages removed from the fat-tree by delivery cycle t come from each of the four subtrees shown in Figure 3-5. At least one of these subtrees provides no more than $m/4$ of the messages. We choose one such subtree and refer to it as the *unfavored* subtree. The other subtrees are referred to as the *favored* subtrees.

Having determined the unfavored subtree given the conditions specified so far, we can complete the construction of M_n. The unfavored subtree contains a copy of the bad message set $M_{n/4}$ for that subtree. Each of the other three subtrees contains $\frac{1}{6}\lambda\sqrt{n}$ messages evenly divided among the processors in the subtree. Now we must prove that M_n meets all of our requirements.

First, we show that M_n is consistent with the input pattern specified for the root switch. To show that the input channels of the root switch of the fat-tree are full through the first t delivery cycles, it suffices to prove that the root channels of the four subtrees are full through this time. The root channel of the unfavored subtree is full by the induction hypothesis (property 2). The root channel of each favored subtree is also full for the first t delivery cycles, since its messages are evenly distributed, its switches are greedy, and t times its root capacity does not exceed the number of messages emanating from it.

We now prove that properties 1, 2, and 3 hold for M_n. The load factor in the favored subtrees is less than λ by construction. The load factor is at most λ in the unfavored subtree by the induction hypothesis (property 1), so the number of messages in the unfavored subtree is at most $\frac{1}{2}\lambda\sqrt{n}$, and the total number of messages in M_n is at most

$$\frac{1}{2}\lambda\sqrt{n} + 3 \cdot \frac{1}{6}\lambda\sqrt{n} = \lambda\sqrt{n}.$$

Thus, the load factor of M_n on the fat-tree is at most λ, and property 1 holds. Property 2 is satisfied for M_n because the root switch is greedy. We have already shown that the input channels of the root switch are full through delivery cycle t, so the root channel is certainly full for the required amount of time. Finally, property 3 holds because after running $t = \frac{1}{3}\lambda$ delivery cycles, only $m/4 = \frac{1}{12}\lambda\sqrt{n}$ messages have been removed from the unfavored subtree. If priority had been given to the unfavored subtree, only $\frac{1}{6}\lambda$ delivery cycles would have been required to remove the $m/4$ messages, So by the induction hypothesis (property 3), an additional $\frac{1}{6}\lambda + \frac{1}{12}\lambda\lg(n/4) - \frac{1}{6}\lambda$ cycles are required to empty the unfavored subtree. If we include the original t cycles, the total number of cycles required to deliver all the messages in M_n is at least $\frac{1}{6}\lambda + \frac{1}{12}\lambda\lg n$. ■

When probabilistic admissible switches are permitted, the proof of Theorem 8 can be extended to show that the expected number of delivery cycles is $\Omega(\lambda \lg n)$. The idea is that at least one of the subtrees in Figure 3-5 must be unfavored with probability at least 1/4. We call one such subtree the *often-unfavored* subtree. The construction of M_n proceeds as before, with the often-unfavored subtrees playing the previous role of the unfavored subtrees. In any particular run of *GREEDY*, we expect 1/4 of the often-unfavored subtrees to be unfavored, so there is a $\Theta(1)$ probability that 1/8 of the often-unfavored subtrees are unfavored (Lemma 3). Thus, the probability is $\Theta(1)$ that $\Omega(\lambda \lg n)$ delivery cycles are required, which means that the expected number of delivery cycles is $\Omega(\lambda \lg n)$.

Although we have shown an unfavorable comparison of *GREEDY* to *RANDOM*, it should be noted that *GREEDY* does achieve the lower bound we proved for routing messages out the root. That is, routing of messages out the root or, more generally, up the tree only, can be accomplished by *GREEDY* in $O(\lambda \lg n)$ delivery cycles. This can be seen by observing that the highest congested channel (closest to the root) must drop at least one level every λ delivery cycles. If one could establish an upper bound of λ times a polylogarithmic factor for the overall problem of greedy routing, it would show that *GREEDY* still has merit despite its inferior performance in comparison to *RANDOM*.

3.4 Routing with Larger Channel Capacities

In this section, we improve the results for on-line routing if each channel c in the fat-tree is sufficiently large, that is if $\mathrm{cap}(c) = \Omega(\lg n)$ Specifically, we can deliver a message set M in $O(\lambda(M))$ delivery cycles with high probability, i. e., we can meet the lower bound to within a constant factor. The better bound is achieved by the algorithm *RANDOM'* shown in Figure 3-6.

Theorem 9 *For any message set M on an n-processor fat-tree with channels of capacity $\Omega(\lg n)$, the probability is at least $1 - O(1/n)$ that RANDOM'*

```
z ← 1
while M ≠ ∅ do
    for each message m ∈ M, choose a random number i_m ∈ {1, 2, ..., z}
    for i ← 1 to z do
        send all messages m such that i_m = i
    endfor
    z ← 2z
endwhile
```

Figure 3-6: The algorithm $RANDOM'$ for routing in a fat-tree with channels of capacity $\Omega(\lg n)$. This algorithm repeatedly doubles a guessed number of delivery cycles, z. For each guess, each message is randomly sent in one of the delivery cycles.

will deliver all the messages of M in $O(\lambda(M))$ delivery cycles, if $\lambda(M)$ is polynomially bounded.

Proof. Let the lower bound on channel size be $a \lg n$, and let n^k be the polynomial bound on the load factor $\lambda(M)$. We consider only the pass of the algorithm when z first exceeds $e2^{(k+2)/a}\lambda(M)$. We ignore previous cycles for the analysis of message routing, except to note that the number of delivery cycles they require is $O(\lambda(M))$.

We first consider a single channel c within a single cycle i from among the z delivery cycles in the pass. Since each message has probability $1/z$ of being sent in cycle i, we can apply Lemma 4 with $p = 1/z$ to conclude that the probability that channel c is congested in cycle i is at most

$$\begin{aligned}
\left(\frac{e\lambda(M)}{z}\right)^{\mathrm{cap}(c)} &\leq 2^{-\frac{k+2}{a}\,\mathrm{cap}(c)} \\
&\leq 2^{-(k+2)\lg n} \\
&= \frac{1}{n^{k+2}} \, .
\end{aligned}$$

Since there are $O(n)$ channels, the probability that there exists a congested channel in cycle i is $O(1/n^{k+1})$. Finally, since there are $z \leq 2e2^{(k+2)/a}\lambda(M) = O(\lambda(M)) = O(n^k)$ cycles, the probability is $O(1/n)$ that there exists a congested channel in any delivery cycle of the pass. ■

3.5 Routing on Butterfly Fat-Trees

This Section describes and analyzes an algorithm for routing on the butterfly fat-tree (Figure 2-2), which runs in time $O(\lambda(M)\lg^3 n)$. It also discusses faster algorithms by Park [76] and Leighton, Maggs, and Rao [61], which depend on stronger switches.

The mechanics of routing on the butterfly fat-tree are somewhat different than on the original. The underlying channel structure for the two fat-trees is the same, but the butterfly fat-tree does not rely on concentrators to make efficient use of the available output wires. Instead, each message sent through the fat-tree randomly chooses which parent to go to next (based on random bits embedded in its address field) until it reaches the apex of its path, and then it takes the unique path downward to its destination. This strategy guarantees that for any given channel through which a message must pass, the message has an equal likelihood of picking any wire in the channel.

As noted earlier, delivery cycles in the butterfly fat-tree require less time than in the ordinary fat-tree, but the algorithm presented here requires more delivery cycles than the algorithms presented in previous sections. In the butterfly fat-tree, the time to perform a delivery cycle is $O(\lg n)$ since the switches have constant depth. The algorithm described and analyzed here requires $O(\lambda(M)\lg^2 n)$ delivery cycles.

The routing algorithm is a modification of the algorithm *RANDOM'*. We simply surround lines 3–6 with a loop that executes these lines $(k+1)\lg n$ times, where $|M| = O(n^k)$.

The proof that the algorithm works applies the analysis from Section 3.2 to individual wires, treating them as channels of capacity 1. Consider a wire w traversed by a message in a p-subset M' of M, and consider the channel c that contains the wire. For any other message in M, the probability is $p/cap(c)$ that the message is directed to wire w when the message set M' is sent. Thus, the probability that w is congested is at most $B(1, \text{load}(M,c), p/\text{cap}(c)) \leq ep\lambda(M)$, and an analog to Lemma 4 holds because the capacity of w is 1. Lemma 5, which says that the probability is $\frac{1}{2}p$ that a given message of M is delivered when a p-subset of M is sent, also holds if the congestion parameter r

is chosen to be $\Theta(\lg n)$.

We can now prove a bound of $O(\lambda(M)\lg^2 n)$ on the number of delivery cycles required by the algorithm to deliver all the messages in M. It suffices to show that with high probability, all the messages in M get routed when the variable z in the algorithm reaches $\Theta(\lambda(M)\lg n)$. When $z \geq r\lambda(M) = \Theta(\lambda(M)\lg n)$, any given message m is sent once during a single pass through lines 3–6, and the probability that the message is not delivered on that pass is at most $\frac{1}{2}$. Thus, the probability that m is not delivered on any of the $(k+1)\lg n$ passes through lines 3–6 is at most $1/n^{k+1}$. Since the number of messages in M is $O(n^k)$, the probability is $O(1/n)$ that a message exists which is not routed by the time z reaches $r\lambda(M)$.

Park [76] and Leighton, Maggs, and Rao [61] have obtained more efficient routing algorithms under different communications models. That is, they consider the effects of stronger switches and, in the latter case, packet routing instead of circuit switching. Park's algorithm has the advantage of being deterministic, and it still runs in time proportional to $\lambda \lg^2 n$ plus a term not exceeding $\lg^3 n$. It does require switches capable of performing arithmetic operations on $\lg n$-bit numbers. Though this increases the component count for fat-trees, it does not increase the area required for fat-trees with appropriately chosen channel capacities as described in Chapter 4. The scheme of Leighton, Maggs, and Rao leads to a similar situation in terms of component count and area requirements, but the running time is further improved by using a packet routing approach. The approach is based on ideas of Ranade [82] and involves assigning each message a random "key" which is used to establish priorities of messages during routing. The result is an algorithm which works in time $O(\lambda \lg n + \lg^2 n)$ in the bit model.

Chapter 4

Tailoring Fat-Trees to Serve as Universal Networks

This chapter shows how to tailor the basic fat-tree architecture to serve as an area-universal network. To achieve universality, we specify the channel capacities in such a way as to keep the area modest while providing enough interprocessor bandwidth to efficiently simulate the communications patterns of competing networks. Based on the discussion in Chapters 2 and 3, we need a choice of channel capacities which yields small area and a good bound on the load factor induced by message sets of competing networks. This chapter provides appropriate channel capacities and gives the universality results which follow.

In detailing the universality of fat-trees, this chapter will delve into certain issues more deeply than earlier works ([44, 61, 62]). It explicitly considers size (amount of attached memory) of processors comprising the networks being compared. One reason processor size is important is that processors in a fat-tree of area A must be of size $\Omega(\lg A)$ in order to address all the other processors.[1] Also, by using a denser packing of fat-tree processors than in [44, 62], artificial restrictions on the number of processors in a competing

[1] The requirement is $\Omega(\lg n)$, where n is the number of processors, implying $\Omega(\lg A)$ unless the number of processors in the fat-tree is too small to achieve any reasonable simulation.

network are removed. In addition, this chapter shows that by superposing hierarchical mesh connections on a fat-tree, creating a *fat-pyramid* network, it is generally possible to discard assumptions of unit wire delay. Also, comparisons between fat-trees and networks of different total area are considered. Finally, it is shown that all the simulation results remain valid even without the assumption that computation and communication of competing networks proceed in separate phases to be simulated separately.

The focus in this chapter is on the choice of universal network and the mapping of other networks to it. We obtain results about how good a network can be at facilitating this mapping and generally leave open the possibility of applying various routing approaches, but the results are occasionally specified concretely for specific routing algorithms.

Most of the universality results in this chapter are expressed in terms of the running time of the message routing algorithm which is selected. Thus, it is possible to see the effect which would result from improved routing algorithms as well as the current best known results. As seen in Chapter 3, the load factor λ is a lower bound on the time required to deliver a set of messages, and the routing algorithms which have been demonstrated for fat-trees obtain (high probability) upper bounds on the delivery time δ in the form

$$\delta(\lambda, n) = f(n)\lambda + g(n) \ ,$$

where f and g are small polylogarithmic functions[2] of the number of processors, n. (Note that this formulation ensures that $\delta(O(\lambda), n) = O(\delta(\lambda, n))$; this fact will be used several times in the following sections.) Of particular interest is the case where n does not exceed 2^λ, implying that the delivery time is just a small polynomial function of λ. We will use δ with one argument to represent this case, i.e.,

$$\delta(\lambda) = \delta(\lambda, 2^\lambda) \ .$$

[2]That is, these functions are upper bounded by a polynomial in the logarithm of the argument. Thus, functions such as $\lg^2 n$ or $\lg^3 n \lg\lg n$ qualify.

In the case of the routing algorithms discussed in detail in Chapter 3 for the basic fat-trees with very simple switches, $\delta(\lg n)$ is $\lg^3 n \lg\lg n$ in the general case and $\lg^3 n$ in the case that the channel capacities are all $\Omega(\lg n)$. Chapter 3 also gave an algorithm for the butterfly fat-tree with $\delta(\lg n) = \lg^4 n$, and the algorithms of Park [76] and Leighton, Maggs, and Rao [61], for butterfly fat-trees with more complicated switches, yield $\delta(\lg n)$ equal to $\lg^3 n$ and $\lg^2 n$, respectively.

Leighton, Maggs, and Rao [61] use their routing algorithm to show that an appropriate universal network of area $\Theta(A)$ requires only $O(\lg^2 A)$ slowdown (in bit-times) to simulate any network of area A and the same type of processors, without any restriction on the number of processors. (Note that this result has been translated from $O(\lg A)$ slowdown in the word model, since this thesis generally expresses times in bit-times.)

For the simplest situation of comparing networks built out of the same (small) processors, the bounds demonstrated here match those of Leighton, Maggs, and Rao [61], though the network used here remains entirely within the fat-tree framework, which allows a more uniform routing strategy. Also, here we go further by taking explicit note of the fact that fat-tree processors must be of size $\Omega(\lg A)$ in order to address all the other processors and that we may wish to consider networks with even larger processors. We make explicit the outcome of comparison between networks of larger and/or different processors and consider the question of best processor size for a universal network.

This chapter shows that an appropriate universal network of area $\Theta(A)$ built from processors of size $\lg A$ requires only $O(\lg^2 A)$ slowdown *in bit-times* (or $O(\lg^3 A)$ with simple switches) to simulate any network of area A, without any restriction on processor size or number of processors in the competing network. Furthermore, the universal network can be designed so that any message traversing a path of length d in the competing network need follow a path of only $O(d + \lg A)$ length in the universal network. Thus, the results are almost entirely insensitive to removal of the unit wire delay assumption used in previous work.

For convenience, we will use the following terminology henceforth:

Definition: We say that network B can Δ-simulate network A if, for any t, the operations performed by network A in time t can be performed by network B in time $t\Delta$.

The rest of this chapter is organized as follows. Section 4.1 shows that an appropriate choice of channel capacities yields a universal fat-tree with processors packed as densely as possible (linear area for unit-size processors). Section 4.2 shows how stronger universality results may be obtained by generalizing to comparison of networks which may differ in processor size. Section 4.3 shows that the same results can usually be obtained even without the assumption of unit wire delay. Section 4.4 analyzes the ability of a universal network to simulate other networks which use more total area but the same processor area. Section 4.5 shows that the assumption of global synchronization of competing networks can be removed, and Section 4.6 contains concluding remarks.

4.1 A Linear-Size Fat-Tree

This section considers comparisons between networks built out of the same processors. It begins by constructing a fat-tree on unit-size processors which occupies area linear in the number of processors. With processors packed so densely, a very simple one-to-one mapping of a competing network's processors to those of the fat-tree ensures that message sets delivered in unit time by a competing network of area A have $O(\lg A)$ load factor in the fat-tree. Fat-tree routing mechanisms require processors of size at least logarithmic in the number of processors (in order to address all other processors), but by simply scaling the measure of area everywhere, the load factor result for unit-size processors leads to a valid simulation result for networks built out of the same processors of size $\Omega(\lg A)$. Nonetheless, we will explicitly introduce processor area since it yields an improved bound for very large processors and prepares the way for comparisons between networks built from different processors.

Specifying the universal fat-tree requires making an appropriate choice of the channel capacity function $c(p)$, which denotes the capacity of a fat-tree channel on top of a subtree of p processors. This section will show that the best construction may not be modular, i.e., the channel capacity on top of a subtree of the network may depend on the total size to which the network is to be built rather than depending just on the size of the subtree. It will be shown that modularity can be obtained using processors of size $\lg^2 A$ to build a fat-tree of area A, but we will see in Section 4.2 that this may increase the cost of simulating networks with processors of a different size if fat-tree routing strategies are developed which come closer to the lower bound on routing time.

Observe first that if we let $c(p) = \left\lceil \sqrt{p}/\lg A \right\rceil$, we can build a fat tree of A unit-size processors in area $\Theta(A)$. The area can be derived by solving a recurrence relation for the side length $S(n)$ of a fat-tree on n processors in the H-tree layout style of Figure 2-1. We have

$$S(n) \quad = \quad 2S(n/4) + c(n) \ ,$$

with solution

$$S(n) = \sqrt{n}S(1) + \sum_{i=0}^{\log_4 n - 1} 2^i c(n/4^i) \ . \tag{4.1}$$

Substituting for $S(1) = 1$ and $c(n/4^i) \leq 1 + \sqrt{n/4^i}/\lg A$, we obtain

$$S(n) \quad \leq \quad 2\sqrt{n} + \sqrt{n}(\log_4 n)/\lg A \ ,$$

and $S(A) = \Theta(\sqrt{A})$.

Now we are prepared to present a result which strengthens the main universality result in [44] (translated to two dimensions) in that the restriction on the number of processors in the competing network is removed, but the proof here is actually more direct. Rather than using the ideas for balancing decomposition trees developed by Bhatt and Leighton [13] and Leiserson [62], we can consider a straightforward geometric mapping of competing networks to fat-trees. (This idea was actually introduced in [44] but only in the case of a fat-tree simulating networks of slightly smaller area.)

Lemma 10 *Consider networks with unit-sized processors, and let $\mathcal{R}$ be the set of all networks of area A. Then, there exists a fat-tree F of area $\Theta(A)$ such that any message set delivered in unit time by a network in $\mathcal{R}$ induces a load factor of $O(\lg A)$ on F.*

Proof. We use the channel capacities $c(p) = \left\lceil \sqrt{p}/\lg A \right\rceil$ to build a fat-tree of $n = A$ processors, which we have shown requires area $\Theta(A)$. Then we can just recursively bisect any $R \in \mathcal{R}$ in the straightforward geometric fashion, cutting nonsquare pieces in the shorter direction, until we have A pieces. There can be at most one processor in each of the pieces of the recursive bisection, and these processors can be mapped to F so that the recursive bisection of R matches the obvious recursive bisection of F. Then the perimeter of a piece of R corresponding to a subtree of $n/2^l$ processors in F is $O(\sqrt{A}/2^{l/2})$. Thus, for a channel at level l of the decomposition, the load factor of messages generated in unit time is

$$\begin{aligned}
\lambda &= O((\sqrt{A}/2^{l/2})/c(n/2^l)) \qquad (4.2)\\
&= O((\sqrt{A}/2^{l/2})/(\sqrt{A/2^l}/\lg A))\\
&= O(\lg A)\ .
\end{aligned}$$

■

It is actually quite easy to generalize the above results to processors of size α. For simplicity, we assume that processors are square, i.e., $\sqrt{\alpha}$ by $\sqrt{\alpha}$. By building a fat-tree on $n = A/\alpha$ processors with channel capacity function $c(p) = \left\lceil \sqrt{p\alpha}/\lg(A/\alpha) \right\rceil$, we obtain area A and load factor $O(\lg(A/\alpha))$. This can be seen by simply making the correct substitutions into Equations 4.1 and 4.2.

We can also make the channel capacities slightly larger at the lower levels of the tree than was just indicated. (Such a change is required by some of the routing algorithms discussed in Chapter 3.) Specifically, if we increase channel capacities to $\left\lceil\sqrt{\alpha}+\sqrt{p\alpha}/\lg(A/\alpha)\right\rceil$, the asymptotic area of the fat-tree does not increase, as can be seen by substituting into Equation 4.1. The load factor bound remains valid, of course, when the channel capacities are increased.

Using our bound on the load factor, we can now state a simulation result in terms of the running time $\delta(\cdot)$ of the message delivery algorithm.

Theorem 11 *Consider networks with processors of size $\alpha = \Omega(\lg A)$, and let $\mathcal{R}$ be the set of all networks of area A. Then, there exists a fat-tree F of area $\Theta(A)$ which can $O(\delta(\lg(A/\alpha)))$-simulate any network in $\mathcal{R}$.*

Proof. We have seen that processors can be mapped one-to-one, and the load factor for each message set of a competing network in $\mathcal{R}$ is $O(\lg(A/\alpha))$. Thus, the delivery time for each message set is $O(\delta(\lg(A/\alpha)))$. ■

Using the routing scheme of Leighton, Maggs, and Rao [61], where $\delta(\lg n) = \lg^2 n$, yields the following corollary:

Corollary 12 *Consider networks with processors of size $\alpha = \Omega(\lg A)$, and let $\mathcal{R}$ be the set of all networks of area A. Then, there exists a fat-tree F of area $\Theta(A)$ which can $O(\lg^2(A/\alpha))$-simulate any network in $\mathcal{R}$.*

This result is similar to a result given in [61] using a fat-tree with the lower levels replaced by meshes. By using the construction here, it is possible to instead retain a uniform routing strategy throughout all levels of the tree. The idea of adding mesh connections to a fat-tree is, however, a useful one when wire delay concerns are addressed, as we shall see in Section 4.3.

One final note is in order. When $\alpha = \Theta(\lg^2 A)$, we can achieve a modular design; we can use $c(p) = \left\lceil\sqrt{p}\right\rceil$, or rounding slightly differently, capacities which double at every other level as in Figure 2-2. These channel capacities are not adequate to justify the use of the routing algorithm presented in Section 3.4, where channels of capacity $\Omega(\lg n)$ were required, but other routing algorithms remain valid.

4.2 Different Sized Processors

In this section, we consider the possibility of comparing a universal network with processors of one size to other networks with processors of different size. In doing so, we must establish correspondences of single processors in one network with multiple processors in another network. We consider both many-to-one and one-to-many mappings of a competing networks' processors to those of a universal network. The combined observations show that a universal network designed to simulate networks of arbitrary processor size is best built with processors of size α in the range $\lg A \leq \alpha \leq \delta(\lg A)$.

In order to compare parallel machines within the framework of this chapter, we must place some limitations on how we compare machines built from different processors. Rather than get involved in such issues as RISC vs. CISC or other detailed questions of best design for an individual processor, we assume that the processors of networks to be compared have the same instruction set and are equally well-engineered to provide the same operations at the same cost in time and space. The one difference in processors which remains under consideration is the amount of memory attached. For simplicity, we assume that the size of the memory does not affect the instruction execution time; incorporating a small cost for increasing memory size would cause little change to the results given here.

We consider in turn the two cases of the universal network having processors which are larger or smaller than the processors of the simulated network. Then we consider more general statements independent of the sizes of processors in the competing network. In any case, let α_X represent the area of a processor in routing network X.

First, let us consider a competing network with processors smaller than those from which the universal network is to be built. In this case, we can map the competing network R to the universal fat-tree F as before, except that the decomposition of R stops before we get down to individual processors. Instead, we map α_F/α_R processors of R to each processor of F. The computations performed by this block of processors in time t, excluding any communication with processors outside the block, can be realized in time $O(t\alpha_F/\alpha_R)$ on the processor of F. Meanwhile, the communication between blocks can be accomplished with overhead $O(\delta(\lg(A/\alpha_F)))$, yielding the following generalization of Theorem 11:

Theorem 13 *For any α_R, let $\mathcal{R}$ be the set of all networks of area A built out of processors of area α_R. Then, for any $\alpha_F \geq \max\{\alpha_R, \lg A\}$, there exists a fat-tree of area $\Theta(A)$ built out of processors of area α_F which can $O(\max\{\delta(\lg(A/\alpha_F)), \alpha_F/\alpha_R\})$-simulate any network in $\mathcal{R}$.* ■

Now, let us consider a competing network with processors larger than those from which the universal network is to be built. In this case, we decompose the competing network $R \in \mathcal{R}$ down to individual processors as before, but we assign α_R/α_F processors of the fat-tree F to simulate each processor of R. We divide the memory of a processor of R among the corresponding α_R/α_F processors of F. Then as long as processors of F are large enough to address the α_R/α_F regions of memory, we can treat the flow of data to the different pieces of memory just as we would the communication among a set of processors being simulated. Thus, we can view the operation of R as proceeding on the larger number A/α_F of subdivided processors, yielding an additional generalization to Theorem 11:

Theorem 14 *For any $\alpha_F \geq \lg A$ and $\alpha_R \geq \alpha_F$, there exists a fat-tree of area $\Theta(A)$ built out of processors of area α_F which can $O(\delta(\lg(A/\alpha_F)))$-simulate any network of area A and processors of area α_R.* ■

It is straightforward to generalize Theorems 13 and 14 to situations in which a single simulated network has processors of many different sizes. In that case, we can simply let α_R represent the minimum size of processors in R, and both theorems will still hold, though the proofs change slightly. In fact, we can combine Theorems 13 and 14, since the α_F/α_R term in Theorem 13 becomes irrelevant for $\alpha_R \geq \alpha_F$:

Theorem 15 *For any α_R, let $\mathcal{R}$ be the set of all networks of area A with each processor having area at least α_R. Then, for any $\alpha_F \geq \lg A$, there* *exists a fat-tree of area $\Theta(A)$ built out of processors of area α_F which can $O(\max\{\delta(\lg(A/\alpha_F)), \alpha_F/\alpha_R\})$-simulate any network in $\mathcal{R}$.* ■

Since the overhead in Theorem 15, will never exceed $O(\delta(\lg A))$ for $\alpha_R \geq \alpha_F$, we see that the overhead for a fat-tree simulating a network with larger processors is largely insensitive to the relative size of processors. Thus, if we wish to build a universal fat-tree to simulate networks of unknown processor size, it seems that we do best by making the fat-tree processors small enough that $\delta(\lg A)$ will always dominate α_F/α_R in comparisons against networks with smaller processors. Recalling that fat-tree routing mechanisms require that processors of F should be large enough to address $\Omega(A)$ processors, we are led to choose the processor size to satisfy $\lg A \leq \alpha_F \leq \delta(\lg A)$. Then we have the following corollaries to Theorem 15.

Corollary 16 *There exists a fat-tree of area $\Theta(A)$ which can $O(\delta(\lg A))$-simulate any network of area A and processors of arbitrary area.* ■

Corollary 17 *There exists a fat-tree of area $\Theta(A)$ that can $O(\lg^2 A)$-simulate any network of area A and processors of arbitrary area.* ■

4.3 Handling Non-Unit Wire Delay

In this section we consider the effect of dropping the unit wire delay assumption. The general graph layout framework developed by Bhatt and Leighton [13] shows that there is enough room in our fat-tree layouts to build sufficiently large drivers for each wire to keep the wire delay constant in the capacitive model. This section shows that even if this constant switching time is not the dominant determiner of wire delay, the bounds on simulation time shown in earlier sections can almost always still be obtained. It introduces a network referred to as a *fat-pyramid*, which is obtained by augmenting a fat-tree with hierarchical mesh connections. A routing path of length d in a competing network of area A corresponds to a path of length $O(d + \lg A)$ in the fat-pyramid, which generally implies that asymptotic simulation overhead is no worse than in the unit wire delay case. Some of the ideas in this section were suggested by Charles Leiserson and Tom Cormen of MIT.

It should be noted that it is reasonable to assume wire delay to be no worse than linear in wire length, since repeaters (extra switches) can always be used to reduce delay to linear. Linear wire delay would be the correct model if technology could be improved to the point where only speed of light limitations constrain the time to switch a length of wire. Then, the measure of unit time would be much smaller, but linear wire delay would be required of any competing network.

It is also helpful to assume a mild "regularity" condition on the wire delay function. (Similar regularity conditions are used elsewhere in the literature (e. g., [11, 20, 63],[2, p. 280]) in order to obtain results about large classes of functions.) Specifically, let $w(d)$ denote the time required to transmit a bit along a wire of length d; then we seek two properties for the function w. First, w should be nondecreasing, and second it should satisfy the following condition:

Definition: A function w is said to satisfy Condition C1 if there exists a constant c such that

$$\frac{w(d+x)}{w(d)} \leq \frac{\lg d + x}{\lg d}$$

for all $x \geq 0$ and $d \geq c$.

It should be noted that Condition C1 is satisfied by most functions likely to be of interest in the context of wire delay. For example, it is satisfied by all functions of the form $cn^q \lg^k n$ for constants c, q, and k such that either $q < 1$, or $q = 1$ and $k \leq 0$. One way to see that all of these functions satisfy Condition C1, is to observe that they satisfy a simpler regularity condition C2, which implies C1.

Definition: A function w is said to satisfy Condition C2 if there exists a constant c such that

$$\frac{w(d+x)}{w(d)} \leq \frac{d+x}{d}$$

for all $x \geq 0$ and $d \geq c$.

Condition C2 implies condition C1 because $1 + x/d \leq 1 + x/\lg d$ for any $x \geq 0$ and $d > 0$.

(Without changing the asymptotic results given below, we can actually weaken conditions C1 and C2 in order to admit an even larger class of functions than already mentioned. Specifically we could define conditions C1 and C2 to be that the old conditions are satisfied to within a constant factor. Then the conditions are satisfied by any function w satisfying $c_1 n^q \lg^k n \leq w(n) \leq c_2 n^q \lg^k n$ for sufficiently large n, with q and k as before and positive constants c_1 and c_2.)

To construct the fat-pyramid network and obtain results independent of wire delay, we must begin with a regular layout of a fat-tree, that is, one in which the components at any given level of the tree are regularly spaced throughout the layout. We can produce such a layout from a butterfly fat-tree by using the "fold and squash" technique of Bhatt and Leighton [13, pp. 325–326] and Thompson [96, pp. 36–38]. Then with only a constant expansion in area, we can superpose a mesh on each level of components in the fat-tree. For ease of illustration, the superposition of these hierarchical mesh connections is shown on a standard layout of the butterfly fat-tree in Figure 4-1. The result of the fold and squash transformation is illustrated in Figure 4-2 with inter-level connections omitted but switches labeled according to the number of levels from the top of the tree. (The fold and squash technique appears again in Chapter 5, and the illustrations there can be used to visualize the transformation of the inter-level connections.) In general, we must stop folding when we reach a level at which the channel capacities stop decreasing sufficiently (by a factor of 2 at every level in the underlying 4-ary tree), so we may be left with H-trees near the leaves. In any case, we use a fat-tree with processors of area at most $\lg^2 A$ in accordance with the results of Section 4.2, and the area of the H-tree blocks is $\lg^2 A$.

The choice of the name "fat-pyramid" for the network just introduced stems from the observation that if all channel capacities were equal to one, the result would be a network

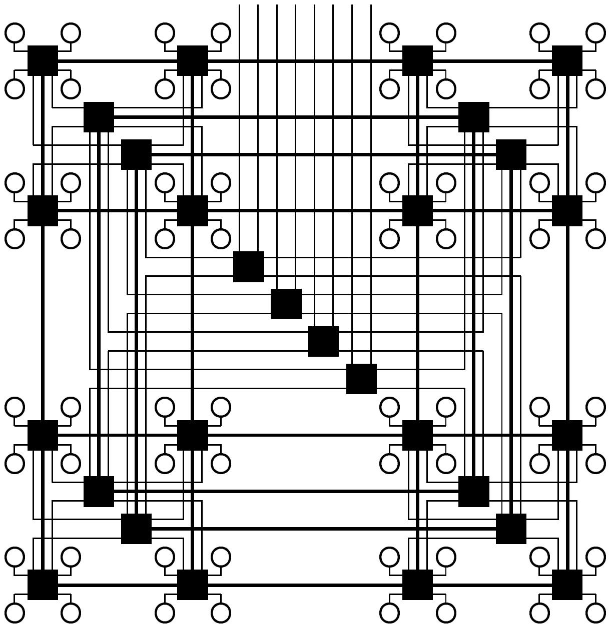

Figure 4-1: A fat-pyramid. This network is obtained by superposing hierarchical mesh connections on a butterfly fat-tree. The original fat-tree connections are represented by thin lines and the mesh connections by thick lines.

which has been called the “pyramid” by Tanimoto (and earlier a “recognition cone” by Uhr) [94, p. 3]. The addition of mesh connections to the fat-tree is also similar to the introduction of “brother” connections in trees to obtain the X-Tree network [32, 89].

Messages in the the fat-pyramid are routed over the same paths as in the fat-tree, except that we allow each message to take one shortcut via one or two of the new mesh edges. More precisely the routing path is formed by going up tree edges until a switch is reached that is adjacent horizontally, vertically, or diagonally in the mesh at that level to a switch from which the destination can be reached by going down tree edges. As long as the mesh edges are of sufficiently large constant capacity, all of the existing routing algorithms work as well as before; the shortcuts do not cause any extra messages to go through any of the tree edges, and the messages which go through any mesh edge are only those which would have gone up the tree from a constant number of nearby switches.

A key property of the layout in Figure 4-2 is that the wires connected to a switch l

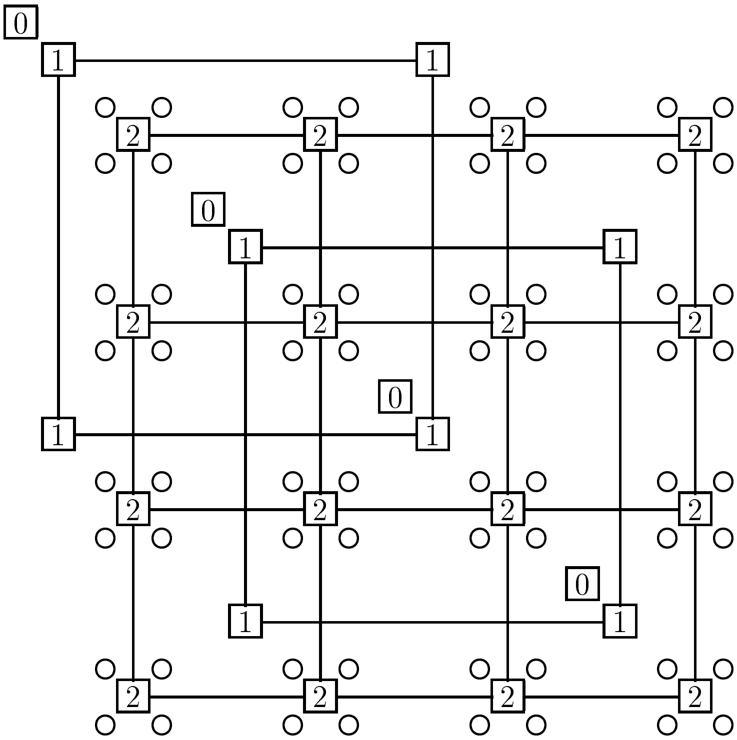

Figure 4-2: A regular layout of the fat-pyramid, but with the original fat-tree connections removed for ease of illustration. Each switch is numbered with its level from the top of the tree so that this figure can be compared with Figure 4-1. The network shown here allows good results to be obtained without the unit wire delay assumption.

levels from the bottom of the tree [3] are of length $O(2^l \lg A)$, where A is the area of the fat-tree. To see this, observe that the H-tree blocks near the leaves have side length $\lg A$, and the upper levels can be embedded in a tree of meshes graph of $O(\lg A)$ levels in such a way that each edge connected to a switch l levels up is mapped to at most $O(2^l)$ edges in the tree of meshes. The fold and squash layout of the tree of meshes given in [13] has maximum edge length $O(\lg A)$, and there is at most $\lg A$ extra length caused by the H-tree blocks in our layout. Thus, the length of an edge connected to a switch l levels up in the folded and squashed fat-pyramid is $O(2^l \lg A)$.

Now we can show that the mapping of competing networks to a universal fat-pyramid (having channel capacities as in a universal fat-tree) does not stretch any wires by very much.

> **Theorem 18** *Let R be a network occupying a square of area A. Then, R can be mapped to a fat-pyramid F of area $\Theta(A)$ so that any message following a path of length d in R travels only $O(d + \lg A)$ distance in F.*

Proof. Observe first that if we use the straightforward mapping of R to F, processors separated by distance d in R are at most $\lceil d/\lg A \rceil$ H-tree blocks apart when mapped to F. Since four adjacent subtrees on $(\lceil d/\lg A \rceil)^2$ H-tree blocks must suffice to cover such a pair of processors, the routing path connecting these processors needs only to go up $\lg(\lceil d/\lg A \rceil)$ levels and use two mesh edges. Since any wire connected to a switch l levels up the tree is of length $O(2^l \lg A)$, the length of the routing path connecting processors at distance d in R is $O(d + \lg A)$. ■

The above result can be used to almost always insure that asymptotic simulation time does not degrade. Clearly, simulation time does not degrade if the competing network operates in separate phases of communication and computation and frequently produces messages which must travel over distance at least $\Omega(\lg A)$. But we can obtain more general results by using the regularity conditions assumed for the wire delay function w

[3] Though discussing the fat-pyramid, we will sometimes make reference to locations in the network relative to the underlying tree structure prior to superposition of the mesh connections.

and by imposing some new rules of operation on the universal network. We henceforth exempt from consideration the packet routing scheme of Leighton, Maggs, and Rao [61], but an analysis similar to the discussion below appears to be applicable to their scheme as well [69].

First we show that when the unit wire delay assumption is dropped, the delivery overhead is at most doubled for a single set M of messages which reach their destinations during a period of unit time in the competing network. This result is shown by comparing the change in time required to deliver the messages in the universal network to the change in time required in the competing network. With regard to the universal network, we need only consider the change in the time to perform each delivery cycle, since the number of required delivery cycles is not changed by the change in wire delay. If we let d be the length of the longest routing path in R for messages in M, then the change in required routing time in R is at least $w(d)$. That is previously, we assumed that all the messages were delivered in unit time, but now we know that the routing time must certainly be at least $w(d)$. In the universal network, we know from Theorem 18, that the time required for a delivery cycle is at most the product of $w(d+\lg A)$, the number of fat-tree switches in the longest routing path ($\lg d$), and the depth of the switches. In the unit wire delay case, we assumed a time equal to the height $\lg A$ of the tree multiplied by the depth of the switches, so the change in time in the universal network is a factor of $w(d+\lg A)\lg d/\lg A$. Thus the change in delivery overhead, which we denote by $\Delta\delta$, is as follows:

$$\begin{aligned}\Delta\delta &\leq \frac{w(d+\lg A)\lg d}{w(d)\lg A}\\ &\leq \frac{(\lg d+\lg A)\lg d}{\lg d\lg A}\\ &\leq 2\ ,\end{aligned}$$

where the second line follows from regularity condition C1.

The result we have just shown is not quite sufficient to establish an on-line simulation of other networks by fat-pyramids because it suggests the use of delivery cycles of differing length according to the longest routing path for the message set being delivered. Individual processors cannot know how long to wait for each delivery cycle to complete when they have no global knowledge of the message set. Nonetheless, it is possible to achieve an on-line simulation by eliminating the clear demarcation between delivery cycles and letting processors send new messages when they are ready regardless of the status of messages sent from other processors. This approach raises an issue of fairness; that is, how do we prevent some message from being "locked out" for a long time while other messages are continually injected into the network, congesting channels needed by the unfortunate message? The answer is basically to time-stamp each message with its time of generation and use these time-stamps to determine priority in the switches of the fat-pyramid. Since the fat-tree routing algorithms which have been presented made no assumptions about how messages were prioritized as they passed through the switches, the routing algorithms and running times remain valid with the time-stamp priority scheme. The only detail that remains is to clarify what happens when a message arrives at a switch with a full output channel but has a higher priority than some message which is in the process of being delivered through that switch. It is necessary in this case that the input messages be buffered or else that the high priority message can actually abort the delivery of a lower priority message.

4.4 Simulating Larger Networks

This section obtains upper bounds on the time required by a universal fat-tree to simulate networks that occupy more area but have the same amount of area devoted to processors. The reason for the latter restriction is that for any significant difference in memory, there are computations which can be performed in the larger amount of memory space but not in the smaller amount of memory space. Rather than placing restrictions on the type of computation, it is probably more meaningful to look at restrictions on the way that space is allocated. That is, if the larger network uses the same amount of processor area (including memory) and simply uses more interconnect area, then we can make meaningful comparisons between the networks. As would be expected, simulation difficulty increases as the area of the competing network does, but only up to a certain threshold beyond which extra area does not help the competition.

As we open up the issue of restricting the processor area used by competing networks, it may seem natural to ask about situations in which the competing network has less processor area than is allowed for the universal network. Indeed, we could have considered this question earlier when comparing networks of the same total area. But when the processor area of competing networks is so restricted, the best results are obtained by tailoring the universal network to the particular mix of processor and interconnect area, with the most difficult case occurring when the competing network has no less processor area than the universal network. Thus, the results given so far are worst-case results for simulating networks of essentially the same total area. In this sense, these networks are the best known to build in a given area. Rather than digress to networks tailored to particular mixes of processors and interconnect, we now ask how well the networks discussed so far can do when they are actually matched against networks of larger area but with no greater processor area.

In what follows, we let A_X represent the area of network X. Of necessity, however, we consider only competing networks in which the processor area does not exceed that of our universal fat-tree F. We are, of course, interested in the case where the competing network R has at least as much area as F, i.e., $A_R \geq A_F$. When $A_R \leq A_F$, our earlier results apply. For simplicity, we assume unit-size processors; the other variations discussed in the previous subsection are easily incorporated.

We use the same basic strategy as before for demonstrating universality results; that is, we recursively bisect the competing network and map appropriate pieces to the fat-tree processors. But when the competing network may have greater area than processor area, extra care is required to ensure that the decomposition is balanced; that is, when we bisect the area of the competing network, we must also bisect the competing processors (or pieces of processor area as in Section 4.2). Fortunately, we can invoke the general theory developed by Bhatt and Leighton [13] and, in a fashion that is cleaner for our purposes, by Leiserson [62]. (It is not desirable to use this approach when unnecessary due to a "loss of locality" in the mapping, which destroys the results on nonunit wire delay in Section 4.3.) These results tell us that since the competing network of area A_R has a decomposition using cuts of size $\sqrt{A_R}/2^{l/2}$ at level l, it has a balanced decomposition using cuts of the same size (up to a constant factor). Keeping this fact in mind, we can prove the following theorem:

Theorem 19 *Let $\mathcal{R}$ be the set of networks of total area A_R and processor area A_F. Then a universal fat-tree of area $O(A_F)$ can $O(\delta(\sqrt{A_R/A_F}\lg A_F, A_F))$-simulate any network in $\mathcal{R}$.*

Proof. Using a balanced decomposition for $R \in \mathcal{R}$ as described above, we find that the load factor of a set of messages delivered by R in unit time is

$$O\left(\frac{\sqrt{A_R}/2^{l/2}}{\sqrt{A_F/2^l}/\lg A_F}\right)$$

at level l from the root of the fat-tree. The result follows. ∎

When the area of the competing network is much larger than the area of the universal fat-tree, we can actually do better than is suggested by Theorem 19. When A_R is $\Omega(A_F^2)$, the competing network is limited more by the restriction on processor area than by its total area. This is true because communication out of a piece of network R is limited not only by the perimeter of that piece but also by the perimeter of the processors in the piece. Thus, at level l in the balanced decomposition of R, only $O(A_F/2^l)$ messages can leave a corresponding piece of R in unit time. Dividing by fat-tree channel capacity to determine the load factor, we obtain the following result:

Theorem 20 *A universal fat-tree of area $O(A_F)$ can $O(\delta(\sqrt{A_F}\lg A_F, A_F))$-simulate any network having processor area $O(A_F)$.* ∎

4.5 Asynchronous Networks

We show here that if the competing network does not have processors synchronized into global computation and communication phases, it still suffices to separately bound the time required to simulate computation and the time required to simulate communication. The basic model is that each processor in the competing network executes an arbitrary sequence of intraprocessor instructions and interprocessor communications, with no global coordination among the processors, so that computation and communication may be occurring concurrently throughout the network. (It is necessary that if the competing network performs intraprocessor computation and interprocessor communication in parallel, then the simulating network also has this capability.) The naive approach to handling asynchrony is to repeatedly simulate the delivery of messages received during a period of unit time in the competing network followed by simulation of the intraprocessor instructions performed during that period of time. But this approach ignores the fact that one processor may be simulating many others which perform indivisible instructions of differing duration. A variation of the naive approach yields the desired result, however, as detailed in the lemma below.

In what follows, we will take the perspective of a transformation on instruction streams. That is, we simply show that the operations performed by the competing network could be performed in a reasonable amount of time on the simulating network. If operations have predictable execution times, and programs for the competing network express the dependencies between intraprocessor instructions and interprocessor communications, then it is possible to establish a transformation on programs for an on-line simulation.

In the statement of the lemma, note the distinction between messages reaching their destinations in unit time, where we say nothing about the total delivery time, and intraprocessor computation fully performed in a period of time, where we expect the complete execution of the relevant instructions to have occurred during that period of time.

Lemma 21 *Suppose that the set of messages reaching their destinations in* *network A during any period of unit time can be delivered by network B in time f. Suppose also that for any time t, the intraprocessor computation fully performed by network A in time t can be performed by network B in time tg. Then the complete operation of network A over time t can be simulated by network B in time $t \max\{f, g\}$.*

Proof. The proof is by induction on the time t, and we actually prove a stronger result by requiring that network B respect the order of completion of intraprocessor instructions in A. That is, the instructions executed by each processor in B are executed in order of their completion times in A. This restriction is to be met regardless of any tie breaking rule which is adopted for ordering the instructions by completion time in A. Let us refer to any total ordering of all the instructions which is consistent with (is a superset of) the partial order of completion times in A as a *consistent* ordering.

For the base case, consider the operation of network A in the time interval $[0, 1]$. Within such a short time period, none of the messages delivered could depend on any other computation, nor could the intraprocessor computations depend on any of the messages. Thus it suffices to spend time f delivering the messages delivered by A in $[0, 1]$ and, concurrently, time g to simulate the intraprocessor computations of A completed in $[0, 1]$. Furthermore none of the instructions executed by the processors of B depend on one another, so the result holds for any ordering of the instructions.

Now, we assume that $t \max\{f, g\}$ time suffices to simulate the operation of network A over the time interval $[0, t]$ while respecting any consistent instruction order, and we prove that the corresponding result holds for $t + 1$. Consider the operations performed by network A in the time interval $[0, t + 1]$, and let I be the set of instructions completed by A in $(t, t + 1]$. Also, let $\mathcal{O}$ be any consistent instruction order. We replace I with a modified set of instructions before invoking the induction hypothesis and then show how to transform the resulting simulation into a simulation for the original problem which respects the order $\mathcal{O}$.

Consider replacing each instruction $i \in I$ with an instruction $\mathrm{beg}(i)$, which starts executing in A when i does but ends at time t. Then the induction hypothesis can be invoked on the operation of A during $[0, t]$ with each $i \in I$ replaced by $\mathrm{beg}(i)$ but $\mathcal{O}$ unchanged. The result is a simulation by B in time $t \max\{f, g\}$ with the truncated instructions from I being the last performed by any given processor in B.

Now we can transform the simulation by B into a simulation of the original operation of A by doing little more than replacing each execution in B of $\mathrm{beg}(i)$ with the execution of i. Since no other instructions or messages can depend on instructions in I, there is no harm in delaying the execution of instructions in I in order to substitute i for $\mathrm{beg}(i)$. The extra time required is just the g time which would be required to perform the left over parts of the instructions in I if they were independent instructions. Finally, all that remains is to concurrently deliver the messages which are delivered by A in $(t, t+1]$. Since this can be accomplished in time f, the total simulation time is $t \max\{f, g\} + \max\{g, f\}$.

■

4.6 Remarks

This chapter has shown that a fat-tree network can efficiently simulate any other network built in the same amount of area, without significant additional restrictions. It has further shown that under modest assumptions about differing processors, a parallel machine based on the fat-tree network can efficiently simulate any other parallel machine built in the same area, regardless of the relative processor sizes.

This chapter has also given other important results. First, dropping the unit wire delay assumption does not generally affect the simulation time required by universal networks. This result was shown by introducing a new network, the fat-pyramid, which is obtained by adding hierarchical mesh connections to a fat-tree. Second, given a universal fat-tree, we can obtain bounds on the time to simulate larger networks of the same total processor area. Unfortunately the latter result is not readily extended to the case of nonunit wire delay, due to the use of decomposition trees that are balanced. It is an open question whether or not this extension can be achieved. Perhaps it could be shown that there is a balanced decomposition tree which will not force nearby processors to be mapped too far from each other in the universal fat-pyramid.

The results given in this chapter suggest that when building a universal network of area A, the best processor size to use is $\Theta(\lg A)$, or $\Theta(\lg^2 A)$ if modularity is more important than the ability to simulate networks of very small processors. The apparent lesson for the design of general-purpose parallel machines is that as total memory demands increase, most of the effort should go into expanding the number of processors rather than individual processor size.

A key open question is whether or not the routing algorithms for universal networks can be improved. It is possible that a better fat-tree routing algorithm could yield only $O(\lg A)$ slowdown in bit-times. Furthermore, it might be possible to take greater advantage of the hierarchical mesh connections in the fat-pyramid. If the notion of load factor is generalized to arbitrary networks by considering arbitrary cuts of the network as opposed to fat-tree channels, then network mappings onto the fat-pyramid yield constant load factor. It is possible that a routing scheme more clever than one relying almost exclusively on the tree connections could yield better message delivery times.

Chapter 5

Three-Dimensional Layouts

This chapter provides a simple demonstration that the $n \times n \times n$ tree of meshes graph truncated to l levels can be embedded in a three-dimensional (3D) grid using volume $O(n^3 l^{3/2})$. The maximum wire length in the embedding of the graph is $O(l^{1/2})$. This embedding simplifies the method developed by Leighton and Rosenberg [57] for the 3D embeddings of general graphs and allows the framework of Bhatt and Leighton [13] for 2D graph layout to be extended to 3D. It also leads to good 3D layouts for the 3D meshes of trees graph [58] and 3D versions of the fat-tree networks discussed in this thesis and in earlier works [44, 62]. Most of this chapter is derived from a joint paper with Charles Leiserson of MIT [43].

The problem in VLSI graph layout is to embed a bounded-degree graph in a two-dimensional (2D) grid by mapping vertices of the graph to gridpoints and edges of the graph to edge-disjoint paths in the grid. Divide-and-conquer algorithms for VLSI graph layout were given by Leiserson [63] and Valiant [106]. The 2D tree of meshes graph (see Figure 5-1), introduced by Leighton [56, 59], was used by Bhatt and Leighton [13] to develop a more comprehensive framework for graph layout. Bhatt and Leighton's upper bounds on area and wire length for embedding graphs into a grid are derived in two steps. The first step is to show that any bounded-degree graph can be embedded in an appropriate subgraph of a tree of meshes. The second step is to obtain an efficient layout

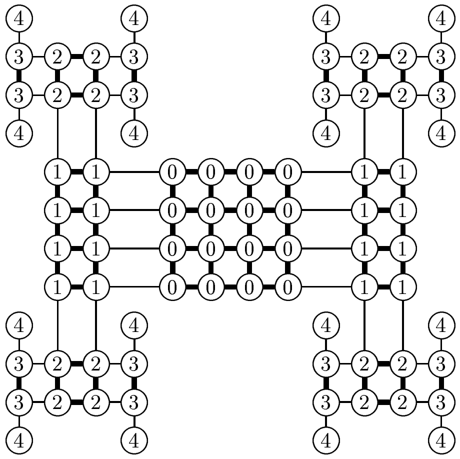

Figure 5-1: The 4×4 tree of meshes $T_{2,4}$ in the H-tree layout style. Each node is labeled with its level in the underlying tree. The "mesh" connections are shown with thick lines, and the "tree" connections are shown with thin lines.

for the tree of meshes.

Upper bounds for three-dimensional (3D) graph layout were derived by Leighton and Rosenberg [57]. Their bounds were derived using somewhat complicated constructions. In this chapter, we obtain their bounds in a simpler fashion by extending Bhatt and Leighton's 2D approach to 3D. The key is a compact layout for the 3D tree of meshes.

A 4×4 tree of meshes is illustrated in Figure 5-1 in the H-tree layout style. The 3D tree of meshes graph is analogous. It is formed by replacing each node of a complete binary tree by a 3D mesh and each edge by several edges which connect a face of a mesh at one level of the tree to a face of a mesh at the next level. In the $n \times n \times n$ tree of meshes, the root of the tree is replaced by an $n \times n \times n$ mesh, and nodes at succeeding levels are replaced by meshes of size $n \times n \times \frac{n}{2}$, then $n \times \frac{n}{2} \times \frac{n}{2}$, then $\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2}$, and so forth until the leaves are replaced by $1 \times 1 \times 1$ meshes. We use the notation $T_{2,n}$ for the $n \times n$ 2D tree of meshes and $T_{3,n}$ for the corresponding 3D graph.

For some of the graph layout upper bounds based on the tree of meshes, it is necessary to consider only the upper levels of the tree of meshes. We will refer to the subgraph of the $n \times n \times n$ tree of meshes consisting of levels $0, 1, \ldots, l$, where $l \leq \frac{3}{2} \lg n$, as the (3D) truncated tree of meshes $T_{3,n}^{(l)}$. The corresponding graph in 2D (with $l \leq 2 \lg n$) will be denoted by $T^{(l)}_{2,n}$.

The area of the 2D tree of meshes can be found by writing a straightforward recurrence for the side length of the H-tree layout. If we let $S(n)$ be the side length of the $n \times n$ tree of meshes, we have $S(1) = 1$, and

$$S(n) \leq 2S(n/2) + O(n) ,$$

which gives $S(n) = O(n \lg n)$ and an area of $O(n^2 \lg^2 n)$. (The matching lower bound is obtained by Leighton [56, 59]; this chapter concentrates on the upper bounds.) Similarly, if we restrict attention to the truncated tree of meshes, $T^{(l)}_{2,n}$, we obtain area $O(n^2 l^2)$

For the 3D tree of meshes, the straightforward recurrence does not suffice to obtain the best bound. If we let $S(n)$ be the side length for the 3D H-tree style layout of the $n \times n \times n$ tree of meshes, we obtain the same $O(n \lg n)$ result for the side length $S(n)$ as in the 2D case above. Cubing the side length yields a volume of $O(n^3 \lg^3 n)$, which is not the best possible bound. Using the method of Leighton and Rosenberg [57], we can obtain the asymptotically optimal volume $O(n^3 \lg^{3/2} n)$ for the $n \times n \times n$ tree of meshes. Their scheme involves solving three simultaneous recurrence relations on the dimensions of the layout.

Our layout for the $n \times n \times n$ tree of meshes presented in Section 5.1 also obtains the asymptotically optimal volume $O(n^3 \lg^{3/2} n)$, but without solving recurrences. Moreover, unlike the scheme of Leighton and Rosenberg, our layout strategy treats the three dimensions symmetrically. Our method is a "fold and squash" layout technique based on ideas of Bhatt and Leighton [13] and Thompson [96] for 2D layout. We show, in particular, that the truncated tree of meshes $T^{(l)}_{3,n}$ can be laid out with volume $O(n^3 l^{3/2})$ and maximum wire length $O(l^{1/2})$. In Section 5.2 we discuss the implications of this result for general 3D graph layout based on decomposition trees. Finally, Section 5.3 discusses variations on the usual decomposition tree framework which are useful for extending the results of Chapter 4 to three dimensions.

5.1 Layout of the 3D Tree of Meshes

In this section we provide a construction showing that the truncated 3D tree of meshes $T_{3,n}^{(l)}$ can be laid out with volume $O(n^3 l^{3/2})$ and maximum wire length $O(l^{1/2})$. This construction is a "fold and squash" technique similar to that used by Bhatt and Leighton [13, pp. 325–6] to produce a layout for the 2D truncated tree of meshes $T_{2,n}^{(l)}$. We begin with a description of such a layout for $T_{2,n}^{(l)}$ in order to lay the ground work for the analogous 3D construction.

The first step in the layout of $T_{2,n}^{(l)}$ is to create a "folded" 3D layout. Starting at the top of the underlying tree and working our way down, we bend each of the tree edges, so that the meshes at any given level are folded naturally into a layer directly above the meshes of the previous level. Another way to view this operation, which will be helpful later, is to think of a reflecting line perpendicular to each set of tree edges in Figure 5-1. At each level, the meshes are reflected across this line and displaced in the third dimension. We will refer to this third dimension as "level," while the other two dimensions, which appear in the final layout, will be referred to as row and column. We will also refer to the row and column grid as the "underlying grid". For every position in the underlying grid, the folded layout gives l points stacked in levels on top of one another.

The second step in the layout of $T_{2,n}^{(l)}$ is to project ("squash") the folded layout onto the plane in the manner of Thompson [96, pp. 36-37], so that the meshes appear as in Figure 5-2. Under this projection, each position of the underlying grid, is expanded into an $l \times l$ square, which we will refer to as a *cubie*, since this term will also be used in three dimensions. The l points stacked on a given position of the underlying grid are mapped into a cubie in such a way that no two points lie in the same row or column. This mapping ensures that the mesh connections from the various levels of the tree do not conflict. By analogy to chess, we may view the mapping of the l points into the cubie as the procurement of a set of l points which are *rookwise independent*. That is, no two rooks (which move in a straight line in one dimension) can take each other when they

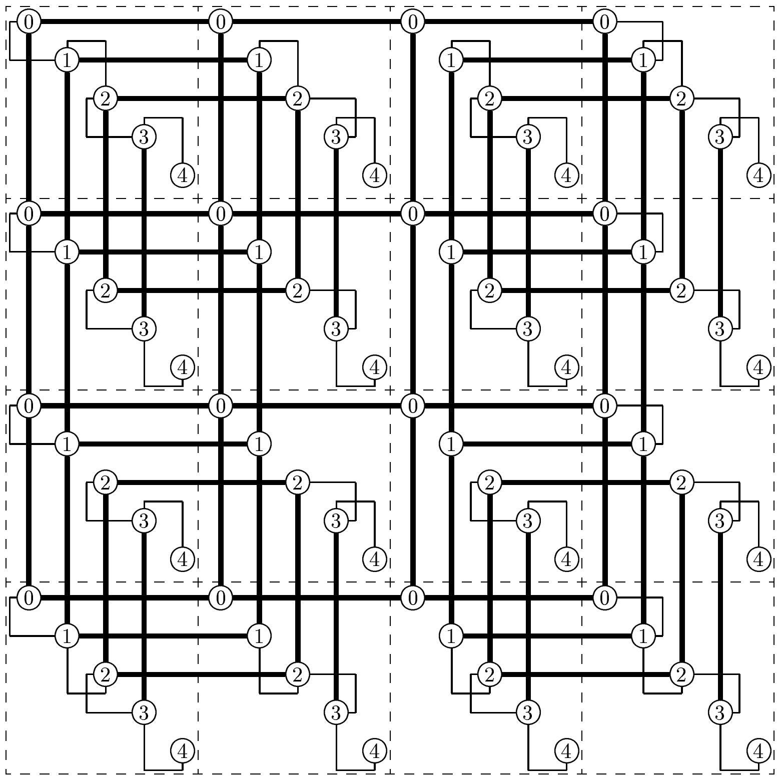

Figure 5-2: The 4×4 tree of meshes $T_{2,4}$ laid out by the fold-and-squash method. Each node is labeled with its level in the underlying tree. The "cubies" are outlined with dashed lines. Mesh edges are shown with thick lines and are routed between cubies. Tree edges are shown with thin lines and are routed within cubies.

are placed in the positions of the cubie used to hold mesh points. In two dimensions, it is straightforward to obtain l rookwise independent positions in an $l \times l$ square, as is shown in Figure 5-2. Since we started with an $n \times n$ underlying grid and have expanded each dimension by a factor of l, the area used so far is $nl \times nl$.

All that remains is to show that the tree connections which run between levels can be routed without increasing the area by much. Leighton and Rosenberg show by case analysis that these connections can be added in the general context of squashing the complete 3D grid into two dimensions [57]. Most of the cases in the general proof actually are not needed when dealing with the particular graph $T_{2,n}^{(l)}$, and, even in general, the expansion need not be as much as Leighton and Rosenberg use. By essentially doubling the number of rows and columns, the tree edges can be routed as shown in Figure 5-2

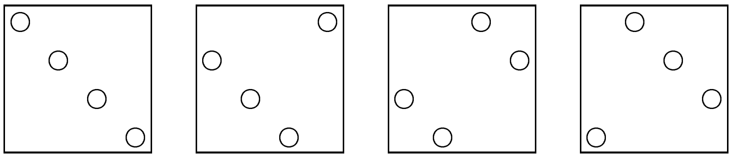

Figure 5-3: Sixteen rookwise independent points in a $4 \times 4 \times 4$ grid, shown in cross sections.

by a method which easily generalizes to three dimensions. Auxiliary rows and columns are added between each pair of existing rows or columns and at the periphery of the grid. Each tree edge attaches to its two endpoints from the direction of the reflecting line which had crossed the edge; the remaining portion of the edge is embedded in the auxiliary row or column which is the first one lying in that direction from both points. Thus, the final layout for $T_{2,n}^{(l)}$ is $O(nl) \times O(nl)$.

The key to extending the 2D layout result to 3D is to obtain an appropriate result on rookwise independence in three dimensions. We first make a precise definition of rookwise independence which maintains the notion of a rook move being a straight rectilinear line.

> **Definition:** A set of points in a grid (of arbitrary dimension) is *rookwise independent* if every two distinct points differ in at least two coordinates.

Figure 5-3 illustrates a rookwise independent set of 16 points in a $4 \times 4 \times 4$ grid by showing four cross sections of the grid.

Now we are ready to present a simple, but important lemma on rookwise independence.

> **Lemma 22** *For any square integer l, there exists a set of l rookwise independent points in a $\sqrt{l} \times \sqrt{l} \times \sqrt{l}$ grid.*

Proof. Let S be the set of l points with the following coordinates:

$$S = \left\{ (i \bmod \sqrt{l}, i+j \bmod \sqrt{l}, j \bmod \sqrt{l}) \mid 0 \le i,j \le \sqrt{l} \right\} .$$

Suppose that two points in S,

$$(i_1 \bmod \sqrt{l}, i_1 + j_1 \bmod \sqrt{l}, j_1 \bmod \sqrt{l}) \text{ and } (i_2 \bmod \sqrt{l}, i_2 + j_2 \bmod \sqrt{l}, j_2 \bmod \sqrt{l}) ,$$

differ in only one coordinate. Whichever two coordinates are equal, the implication is that $i_1 \equiv i_2 \bmod \sqrt{l}$ and $j_1 \equiv j_2 \bmod \sqrt{l}$, i.e., the two points are identical. ■

With Lemma 22 in hand, we can proceed to the main result on the layout of the 3D truncated tree of meshes.

Theorem 23 *The 3D truncated tree of meshes $T^{(l)}_{3,n}$ can be laid out in volume $O(n^3 l^{3/2})$ with maximum wire length $O(l^{1/2})$.*

Proof. We use essentially the same technique as for for $T^{(l)}_{2,n}$ but in a higher dimension. As before, we begin by folding up the levels of the tree. This folding into the fourth dimension is hard to visualize, but we can think of it in terms of reflecting planes analogous to the reflecting lines in the 2D construction. The meshes at each level are reflected across a plane perpendicular to the appropriate tree edges and then displaced in the fourth dimension. We will again refer to this last dimension as level, while the other dimensions, referred to as row, column, and elevation, constitute the underlying grid. As before, we have folded l points onto each point of the underlying grid.

The second step in the analogy is to squash the mesh points of the various levels into actual positions in three-space such that the meshes do not conflict. According to Lemma 22, it suffices to expand each position of the underlying grid into a $\sqrt{l} \times \sqrt{l} \times \sqrt{l}$ cubie. By mapping the l points stacked onto a given position of the underlying grid to rookwise independent positions in the cubie, we guarantee that the various meshes will not conflict.

For the final step, we need to show that little increase in volume is required to perform the routing of tree edges within cubies. We use an approach analogous to that illustrated in Figure 5-2; we add auxiliary planes between the existing planes in each of the three dimensions and at the periphery of the grid. Each tree edge includes *terminal segments* which connect to its endpoints from the direction of the reflecting plane which had crossed the edge. The remaining portion of the edge is routed as a single-bend path in the auxiliary plane which is the nearest one in the relevant direction from the two endpoints. The terminal segments cannot conflict with one another due to the rookwise independent placement of points in the cubie, nor can they conflict with the single-bend paths since they do not lie in an auxiliary plane. Finally, we can prove by contradiction that there does not exist a conflicting pair of single-bend paths in an auxiliary plane. If there did, each of the corresponding edges would have an endpoint lying in the same plane adjacent and parallel to the auxiliary plane, and these points would have to have the same coordinate in one of the dimensions of the auxiliary plane. This would contradict the rookwise independence of the points in a cubie.

Since we started with an $n \times n \times n$ underlying grid and have expanded each dimension by $O(\sqrt{l})$, the volume of the layout is $O(n^3 l^{3/2})$. Also each mesh edge has length $O(l^{1/2})$, while the tree edges within cubies have length at most a constant times the edge length of a cubie, which is again $O(l^{1/2})$. Thus the maximum edge length is $O(l^{1/2})$. ■

5.2 Graph Layout Based on Decomposition Trees

The $O(n^3 l^{3/2})$ volume layout of the 3D truncated tree of meshes $T^{(l)}_{3,n}$ can be used to obtain simpler proofs of the upper bounds obtained by Leighton and Rosenberg [57, p. 807] for 3D layout of bounded-degree graphs. Moreover, this proof technique brings 3D layout into the general framework proposed by Bhatt and Leighton [13].

We present the 3D layout results in terms of the "decomposition tree" formalism from [62], instead of the similar "bifurcator" formalism used by Leighton and Rosenberg.

> **Definition:** A $[w_0, w_1, \ldots, w_r]$ decomposition tree T for a graph $G = (V, E)$ is a binary tree of height r together with an injective mapping from V to the leaves of T such that if T' is a subtree of T at depth i with vertices V', then the cardinality of the set $\{ (u, v) \mid u \in V', v \in V - V' \}$ is at most w_i.

Intuitively, a decomposition tree describes a recursive bisection of a graph. Each parameter w_i gives a bound on the number of edges that connect a subgraph at depth i in the decomposition tree to the rest of the graph. For convenience, we say that G has a (w, α) decomposition tree, where $1 < \alpha \le 2$, if G has a $[w, w/\alpha, w/\alpha^2, \ldots, O(1)]$ decomposition tree.

The following theorem indicates how well a graph can be embedded in a 3D truncated tree of meshes. The theorem is the 3D analog to Bhatt and Leighton's 2D result [13, Thm. 8].

Theorem 24 *There are constants c and k such that every N-node graph with a $(w, 2^{2/3})$ decomposition tree can be embedded in the 3D truncated tree of meshes $T_{3,c\sqrt{w}}^{(3\lg(N/w))}$ with no edge being routed through more than k intermediate meshes.* ∎

Combining Theorem 24 with Theorem 23, we immediately obtain the following version of Leighton and Rosenberg's result [57, Thm. 3.5].

Theorem 25 *Any N-node graph which has a $(w, 2^{2/3})$ decomposition tree can be laid out using volume $O((w\lg(N/w))^{3/2})$ with maximum wire length $O((w\lg(N/w))^{1/2})$.* ∎

There are two specific applications of Theorem 25 that deserve mention. The $n \times n \times n$ mesh of trees graph [58] has an $(n^2, 2^{2/3})$ decomposition tree, and consequently has an $O(n^3 \lg^{3/2} n)$ volume layout. The fat-tree routing network from [62] with n processors and root capacity w, where $n^{2/3} \le w \le n$, has a $(w, 2^{2/3})$ decomposition tree, and consequently can be embedded in volume $O((w\lg(n/w))^{3/2})$. Other fat-tree routing networks, such as the 3D analog of the butterfly fat-tree of [44, Sec. 7], can be similarly embedded.

5.3 Variations of the Usual Decomposition Tree

We have just established bounds on the 3D volume of fat-trees in the style of [62] and [44], but the fat-tree (and fat-pyramid) networks of Chapter 4 use somewhat different channel capacities. In order to obtain appropriate 3D layout bounds for these networks, it is necessary to consider a variation on the basic type of decomposition tree discussed in Section 5.2.

As shown by Leiserson [62], we can restrict attention to balanced decomposition trees, where each cut evenly splits the nodes of the relevant subgraph, but the previous requirement that the decomposition down to isolated nodes is exactly in the form $[w, w/\alpha, w/\alpha^2, \ldots, w/\alpha^{\lg N}]$ places unnecessary restrictions on the size of w relative to the number of nodes, N, in the graph. Since a complete decomposition requires $\lg N$ levels, the framework discussed so far requires that w must be at least $N^{2/3}$ when $\alpha = 2^{2/3}$. But it is natural to consider w of arbitrary size, with $\lg N$ levels in any decomposition tree and cut sizes decreasing by a factor of α only when the cut size will be at least 1.

To extend the results of Chapter 4 to three dimensions, we must handle decomposition trees with $w \leq N^{2/3}$ of the form $[w, w/\alpha, w/\alpha^2, \ldots, 1, \ldots, 1]$ with $\lg N$ levels. For example, the appropriate 3D analog of the channel capacity function used to prove Lemma 10 is $\left\lceil p^{2/3}/\lg V \right\rceil$ on a fat-tree of V processors. This channel capacity function leads to a decomposition tree with cuts of size one before the bottom level of the decomposition. Fortunately, a graph with a $[w, w/\alpha, w/\alpha^2, \ldots, 1, \ldots, 1]$ decomposition of $\lg N$ levels can be embedded in the three-dimensional tree of meshes $T_{3,\sqrt{w}}$ with leaves which are $\frac{N^{1/3}}{\sqrt{w}} \times \frac{N^{1/3}}{\sqrt{w}} \times \frac{N^{1/3}}{\sqrt{w}}$, allowing room for a dense embedding of the lower levels from the graph decomposition. This embedding approach leads to the following theorem.

Theorem 26 *Any N-node graph with a $[w, w/\alpha, w/\alpha^2, \ldots, 1, \ldots, 1]$ balanced decomposition tree with $\alpha = 2^{2/3}$ can be laid out in volume $O(\max\left\{N, (w \lg w)^{3/2}\right\})$ with maximum wire length $O(\max\left\{N^{1/3}, (w \lg w)^{1/2}\right\})$.*

Proof. Consider level $k = \frac{3}{2} \lg w$ in the decomposition, where $w/\alpha^k = 1$. The subgraphs below this level in the decomposition are of $N/2^k = N/w^{3/2}$ nodes and can be packed into a cube of side length $\frac{N^{1/3}}{\sqrt{w}}$. Viewing these cubes as leaves of $T_{3,\sqrt{w}}$ and using the squash and fold layout for the obvious embedding of the graph into the 3D tree of meshes, the graph can be laid out in volume $O([(\sqrt{\lg w} + \frac{N^{1/3}}{\sqrt{w}})\sqrt{w}]^3)$, and the volume result follows according to whether $\sqrt{\lg w}$ or $\frac{N^{1/3}}{\sqrt{w}}$ is larger. The wire length result is also easy to verify.

■

Theorem 26 enables immediate generalization of all the results of Chapter 4 to three dimensions. For example, a volume V fat-tree on V unit size processors can be constructed by using a channel capacity of $\left\lceil p^{2/3}/\lg V \right\rceil$ on top of a subtree of p processors. When a competing network in a cube of volume V is decomposed in the obvious way, the load factor at level l in the decomposition is

$$O((V^{2/3}/2^{2l/3})/((V/2^l)^{2/3}/\lg V)) ,$$

which is $O(\lg V)$.

Part II

Multilayer Channel Routing

Chapter 6

MulCh: A Multilayer Channel Router

Multilayer routing is becoming an important problem in the physical design of integrated circuits as technology evolves towards several layers of metallization. MulCh is a practical channel router accepting specification of an arbitrary number of routing layers. Though several other channel routers for three layers of interconnect have been proposed, the only previously reported practical implementation for an arbitrary number of layers was that of Chameleon [17]. Chameleon is based on a strategy of decomposing the multilayer problem into two and three layer problems in which one of the layers is reserved primarily for vertical wire runs and the other layer(s) for horizontal runs. In some situations, however, it is advantageous to consider also layers that allow the routing of entire nets, using both horizontal and vertical wires. MulCh extends the algorithms of Chameleon in this direction. MulCh can route channels with any number of layers and automatically chooses a good assignment of wiring strategies to the different layers. The algorithms have been devised so that MulCh is expected to always perform at least as well as Chameleon in terms of area occupied by the routing. In test cases, MulCh shows significant improvement over Chameleon in terms of channel width, net length, and number of vias.

This chapter is organized as follows. The first section describes the basic channel routing problem in more detail. The second section reviews previous approaches to multilayer channel routing and explains the division of MulCh into two major parts, the partitioner and the detailed router. The next two sections describe the partitioner and the detailed router, respectively, while the last section presents experimental results.

This chapter is mostly derived from a joint paper with Alex Ishii of MIT and Alberto Sangiovanni-Vincentelli of U. C. Berkeley [42].

6.1 The Channel Routing Problem

Throughout this chapter, we discuss the standard grid-based, channel routing problem. Terminals lie on grid points along two horizontal line segments which delimit the channel. Each terminal is labeled with a net number, and the problem is to connect terminals belonging to the same net, using horizontal and vertical wire segments in a grid of a specified number of layers. (Additionally, some nets may be required to be routed out the left and/or right side of the channel at an unspecified position.) Nets can connect from one layer to another by way of a *contact cut* or *via*; nets cannot intersect one another on the same layer. The primary goal in channel routing is to minimize the channel area. If as is frequently assumed, we cannot extend the channel in the horizontal direction, the objective is to minimize the channel width, the vertical separation between the two lines of terminals. (Variations of the problem, allowing for horizontal shifting of terminals are not considered here.) Secondary goals are to minimize net length and the number of vias. Achieving these secondary goals reduces the time required for signal transmission and increases the probability of fault-free fabrication.

Figure 6-1 shows a channel routing problem and a solution in two layers, where each layer has been reserved for wires running in one direction (referred to as *reserved layer* or *Manhattan* routing). We refer to each of the vertical grid lines as a column, while the horizontal grid lines are referred to as rows or tracks. Though a different convention will be convenient in the next chapter, we assume here that wires must stay unit distance away from the channel boundaries except when connecting vertically to pins. Thus, the

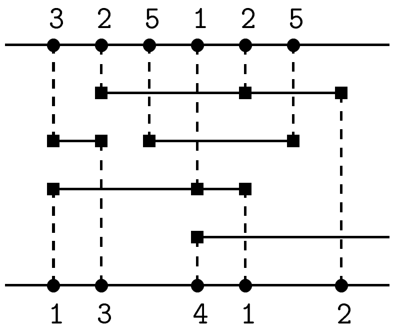

Figure 6-1: A representative channel routing problem in two layers. The horizontal wires (solid) are in one layer and the vertical (dashed) in the other layer. The vias are represented by squares and the terminals by circles.

routing shown in Figure 6-1 achieves the minimum possible channel width when the layers are reserved to individual wiring directions. This optimality follows from the fact that there exist columns which must be crossed by four nets. The convention in this chapter is to refer to a routing such as the one in Figure 6-1 as being of width four, i. e., we count the number of tracks strictly between the lines of terminals.

The lower bound on channel width mentioned in the previous paragraph is one of a few important measures of channel routing difficulty. The bound already alluded to is referred to as the *density* and is generally denoted d. It is the maximum over all columns of the number of nets which must cross the column. A natural variation on this bound will be used in the upcoming discussion of multilayer routing. Another lower bound on channel width in the two-layer Manhattan model is the *flux*, denoted f, which was introduced by Baker, Bhatt, and Leighton [9]. These authors show that any two-layer channel routing problem can be solved in width $2d + O(f)$ (or $d + O(f)$ if all nets have two terminals), and they note that flux appears to be bounded by a small constant for practical problems. This chapter will not make use of flux as a measure of channel routing difficulty; instead it will use an easier to compute measure that has less theoretical justification but is useful in practice. This latter measure is the length of the longest path in the vertical constraint graph (VCG). The VCG is formed by using a node to represent each net and including an edge from net A to net B if a pin for net A appears above a pin for net B in some column of the channel. The length of the longest VCG path is a lower bound on channel width under the Manhattan model with the additional restriction that each net includes only one horizontal wiring segment, i. e., "doglegs" are not allowed. In fact, we will make no restrictions on the use of doglegs, but VCG path length will still be useful as a rough indicator of routing difficulty.

The channel routing problem was first introduced by Hashimoto and Stevens [48] in 1971 and has accumulated an extensive history in the literature. A recent survey has been produced by LaPaugh and Pinter [54]. One of the most important lines of classification for existing channel routing algorithms is the distinction between "provably good" algorithms and "practically good" algorithms. That is, an algorithm that produces excellent results in practice may have a terrible worst-case performance in theory due to pathological instances of the problem unlikely to occur in the real world. On the other hand, an algorithm which has a provably good worst-case performance may virtually always perform worse than a good practical algorithm on "real" problems arising in practice. The ideal combination of provably and practically excellent performance is unlikely to be attained since most variations of the channel routing problem are generally believed to be NP-complete, and, in fact, rigorous NP-completeness proofs have been provided for various two-layer routing styles [55, 93]. This chapter concentrates on a practical, heuristic approach to multilayer channel routing.

6.2 Multilayer Channel Routing

Limited results have been presented for multilayer channel routing. The additional layers of interconnect provide more degrees of freedom, but using this freedom effectively is difficult. Chen and Liu [26] proposed an extension to three layers of the algorithm of Yoshimura and Kuh [109]. Bruell and Sun [23] provided a three layer router based on the greedy algorithm of Rivest and Fiduccia [86]. A three layer algorithm of Heyns [49] actually included a departure from the approach of reserving each layer to a single routing direction; such a departure is an important feature of MulCh. Three and four layer routing algorithms have also been provided by Cong, Wong, and Liu [29]. Enbody and Du [36] have proposed a multilayer algorithm which has been implemented and tested in the three and five layer cases. Additional works on multilayer routing have considered more restrictive models of allowable wiring than are considered here or have been geared more towards the goal of theoretical efficacy rather than practical efficacy [12, 16, 45].[1]

Recently, a multilayer channel router, Chameleon [17], was developed to handle any number of interconnection layers. On various sample problems, Chameleon obtains performance generally superior to the preexisting routers. It can handle channel routing problems with cyclic vertical constraints, with different design rules on different layers and with constraints on the contact locations, i.e., stacked vias can be permitted or disallowed.

The basic approach of Chameleon is to divide the original multilayer problem into essentially independent subproblems which are assigned at most three layers each. Chameleon requires that each layer be reserved primarily for either horizontal wire runs (an H layer) or vertical wire runs (a V layer). The original problem is decomposed by assigning the nets to subproblems, or *groups*, each of which is assigned either one H and one V layer (an HV or VH group) or two H layers and one V layer (an HVH group). Only in the late stages of detailed routing is it possible that wires will digress from this cast. This strategy was inspired by the implicit assumption that the density of the channel is in general much larger than the number of interconnection layers. However, there are situations especially in printed-circuit boards and hybrid circuits where better area utilization can be achieved if sets of nets are routed entirely on one layer.

MulCh is a multilayer channel router that optimizes area utilization by allowing a third type of group, which is assigned a single layer on which wire runs in both the horizontal and vertical directions are permitted (a B layer). This creates obvious complications in that nets assigned to a B layer must not cross if the group is to be routed independently. Use of B layers does, however, give more flexibility and thus may enable a reduction in the area of the route. Possible reductions in total net length and the number

[1] There are also many other references I have omitted which deal with the "knock-knee" model of routing, in which essentially no overlap of wires on different layers is allowed.

of vias are additional benefits.

Like Chameleon, MulCh is composed of two major parts, the partitioner and the detailed router. The partitioner determines the group types used (e.g. HV, HVH, and B) and assigns nets to each of them. The detailed router actually places all the wires in the channel as required to connect pins lying on the same nets. The remainder of this chapter describes first the partitioner and then the detailed router. Finally, some experimental results are presented.

6.3 Partitioning the Problem

The partitioner has two basic tasks. It must choose the group types (e.g., HVH, HV, and B), and it must assign the nets to the groups. In Chameleon, these two tasks are totally separated; the choice of group types depends only on the number of layers (and for nonuniform technology, the mask information). In MulCh, complete separation of these tasks is not possible since the use of B layers introduces uncontrovertible constraints on net assignment. MulCh seeks a good partition by performing a small number of iterations of the two-part process of first choosing group types and then assigning nets. Thus, the choice of group types can be discussed without delving into the details of net assignment, which will be covered afterwards.

For simplicity, attention is restricted to the case of uniform technology with stacked vias allowed. Other technology specifications of the type considered by Chameleon can be handled by MulCh with little additional complication.

6.3.1 Choosing the Group Types

Chameleon is able to choose group types independently from net assignment, because the best results are generally obtained with the largest possible number of H layers. That is, the partitioning strategy and the detailed router are good enough that the resulting channel width is usually very close to the density divided by the number of H layers (the lower bound under strict adherence to the H-V wiring model). Thus the strategy of Chameleon is to use as many HVH groups as possible and up to two HV groups.

MulCh follows the same approach once the number of B layers is set, but it is not as easy to determine how many B layers to use. Using more B layers reduces the standard lower bound on channel width (density divided by the number of layers which permit horizontal routing), but an excessive number of B layers may prevent us from coming as close to this lower bound because of the additional constraints on net assignment and detailed routing.[2]

Taking these two competing considerations into account leads naturally to the following strategy. We start with no B layers, assign nets to the groups, and estimate the area required to complete the routing. Then we add the minimum number of B layers necessary to increase the total number of layers which permit horizontal routing, and again assign nets and estimate the area. We repeat this process as long as the area estimate shows improvement[3] and then settle on the number of B layers which yielded the best area estimate. For example, if the number of layers is seven, we start with group types HVH, HV, HV. Then we try HVH, HVH, B, which increases the number of layers permitting horizontal routing from four to five. If the area has improved, we then try HVH, B, B, B, B, since any intermediate number of B layers will not increase the number of layers permitting horizontal routing. Finally, if the area has again improved, we try seven B layers.

Experience has shown that the strategy just described is almost always adequate to minimize channel width. If one is willing to perform some extra work in search of reduced net length or via count or a long-shot chance of reduced area, it is reasonable

[2]It is interesting to note that if every net has at least one terminal on each of the top and bottom of the channel, then an algorithm of Dagan, Golumbic, and Pinter [31] can be used to determine the number of layers required if *only* B layers are allowed. But if their does exist a net with does not go across the channel, the problem becomes NP-complete [79, Section V.2.2]. Furthermore, the algorithm of Dagan, Golumbic, and Pinter does not provide any obvious insight on what to do when the number of layers available is not sufficient to use only B layers, nor does it indicate how to assign nets to layers so as to minimize channel width if the number of available layers is more than required to use only B layers.

[3]The actual requirement is that the new result be at least as good rather than strictly better.

to try also one and two B layers more than the best number obtained under the scheme above. It should be noted that by starting with zero B layers, we replicate the approach of Chameleon, so we expect that our results will be no worse than those of Chameleon as long as we are reasonably good at estimating area once we have assigned the nets to groups. (The area estimate used for B groups is the actual width as determined by performing the complete route. For HV and HVH groups we base the estimate on density, which is fairly accurate once we have paid attention to other concerns during net assignment.) Also, by starting with few B layers and increasing the number, the running time increases only as we obtain more and more improvement upon Chameleon.

6.3.2 Net Assignment

Once a tentative set of group types has been chosen, net assignment is performed in a manner similar to Chameleon. That is, nets are assigned to groups one at a time, starting with those nets belonging to cycles in the vertical constraint graph (VCG), and then proceeding through the others in order of first appearance when moving across the channel from left to right. Each net is initially added to all the groups, and a cost function for each group is computed. Then the net is removed from all groups except the best.

Two factors prove to be of key importance in estimating the channel width required to route an HV or HVH group. First, the density of the nets in the group divided by the number of H layers in the group is a lower bound on the channel width. Secondly, and less critically, the length of the longest path in the VCG of the group is a lower bound on channel width in the absence of detailed routing strategies which use doglegs or go beyond strict adherence to the H-V wiring model.

For B layers, the considerations are different. First and foremost we must be sure that a proposed assignment of a net to a B layer does not yield a nonplanar collection of nets in that group. This check is easy, because we can efficiently determine at the beginning of the partitioning process what all the pairs of crossing nets are. Then when an assignment of net A is proposed, we can look at each of the nets crossed by net A and see if any of them have been assigned to the group under consideration.

The net crossings can be found by marching around the four boundaries of the channel in a "circle" and utilizing a stack of nets (with some extra operations beyond the normal stack abstraction). Actually we first march around one more time in a preprocessing step to determine for each net where its first and last occurrence are in the marching order. Then in the main pass around the channel, each time we arrive at a net whose current occurrence is not its last, we push it on the stack. When we arrive at a net whose current occurrence is not its first, we also scan down through the stack until we find the previous occurrence of the current net and cut it out from the stack. Each of the nets we pass on the way crosses the current net, and we can associate the crossing information we glean with each of the relevant nets. This algorithm determines all of the net crossings in time linear in the number of net crossings (multiple crossings of the same nets count) plus the number of positions on the channel boundary.

The above discussion of net crossings has glossed over the ordering of the side terminals, which is not fixed in the original input. There is, however, a readily determined ordering (not necessarily unique) of the side terminals which leaves only those net crossings which must exist regardless of the ordering of the side terminals. Such an ordering is determined and fixed for the partitioner, but this will not be taken to impose any restrictions on the detailed router beyond the restrictions imposed by the essential nature of the partition.

Once we know that a set of nets in a B group is planar, density is an important but imperfect measure of channel width. It is still a lower bound on the width required to route the group, but certain bad arrangements of nets may require much larger channel width. For example, the arrangement shown in Figure 6-2 has density 2 but requires a channel width of 4.

The length of the longest VCG path is of even less relevance to the width of a planar group. The example in Figure 6-2 serves also as an illustration of large area with minimal VCG path length. Conversely, it is possible to construct examples with arbitrarily long

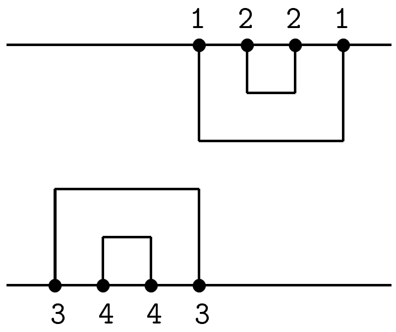

Figure 6-2: This example illustrates low density and VCG path length but high channel width.

VCG path length which can be routed in width two. (Performing such a route does require doglegs, but the routines described in Section 6.4 can compute doglegs for planar routing in polynomial time, in contrast to the NP-complete situation for multilayer groups [93].)

Positively determining the width of a planar group is not straightforward except by essentially performing the complete route. (Chapter 7 describes a more efficient method which was not known when MulCh was implemented.) Though the routing can be performed in polynomial time, experience shows that doing this to evaluate every proposed net assignment causes a large increase in running time on large examples but rarely leads to a reduction in area. In practice, density has proven to be a good estimate of the channel width, but blindly holding to this assumption will cause us to obtain worse results than Chameleon on some bad examples. Thus, density is used as the measure of channel width for planar layers during net assignment, but after completing net assignment for a given number of B layers, the B layers are fully routed in order to estimate channel width. This prevents us from making a major gaffe but avoids doing work which is not usually cost-effective.

Another consideration for B groups is that we would like to avoid including nets which will likely stop us from making many future assignments to B groups. MulCh uses a rather crude penalization on assignment of a net to a B group based on the excess over average of the number of nets it crosses.

Chameleon's cost function for evaluating the assignment of a net to a particular partition group involves variously weighted penalty terms based on the notions discussed for two and three layer groups, but MulCh obtains good results using a substantially simpler synthesis of the measures just described. Each HV and HVH group is assigned a cost equal to the maximum of its density and its longest VCG path. Each B group is assigned cost equal to the density plus a penalty of the excess number of nets crossed by the net being assigned. Each net is left in the group which yields the best cost function; ties are broken first on the basis of density, then longest VCG path length, then number of nets in the group.

Several variations on the cost function have been tried. Most lead to improvement on some example problems and poorer results on other problems. One variant approach actually improves slightly more cases than it worsens but at the expense of some complication in the cost function. In this approach, we include a severe penalty on density exceeding the lower bound on channel width indicated by the number of H and B layers. For HV and HVH groups, we add in the length of the longest VCG path, and this completes the cost function for comparisons among HV and HVH groups. Then in order to make comparisons between these groups and the B groups, we add density into the cost function for each group, and we add the penalty on excess net crossings into the cost for the B groups. Though this variation yields results which are on balance slightly better than those reported in Section 6.5, there are still several cases which have been routed in less area by some other variation of the program. Thus, the variant results may largely represent chance variations rather than intrinsically superior approaches.

6.4 Detailed Routing

The detailed router in MulCh is an enhanced version of the detailed router used in Chameleon. The routing of nets in B groups is performed as a preprocessing step, which routes nets using only their assigned layer. Once all nets in B groups have been routed, the HVH and VH routing routines from Chameleon are used to route all remaining nets. Area on a layer assigned to a B group that is not needed to route nets assigned to the group is available for routing other nets.

Routing of nets assigned to B groups is accomplished with a data-structure called a *plane*. A plane is simply a two-dimensional array, with a number of columns equal to the number of columns in the channel, and a number of rows equal to the number of nets in the B group it is used to route. Each entry corresponds to a 2-D grid point on the layer assigned to the group, and thus a row represents a track which may be needed to route the nets in the group. The assumption that the number of tracks needed is less than the number of nets is valid, since the need for more tracks would imply that the nets in the group are not planar.

While not optimal with respect to worst-case running time, the routines for routing B groups are easy to implement and return optimal width routes for the chosen Manhattan wiring model. Worst case running time is not a major concern, since the routing of B groups is fast in comparison to other groups.

The algorithm for planar routing proceeds in a column-by-column sweep starting from the left edge of an "empty" plane. Nets are routed straight across rows of the plane until a change of course is forced by nets which enter or leave the channel at a given column. Nets entering from the top of the channel are routed down the column as far as room permits, while nets entering from the bottom of the channel may need to push other nets up out of the way. When nets are pushed up, the effect is rippled back through previous columns in order to maintain a valid routing.

The algorithm for routing with the plane can be viewed as wave-like, where the "wave" moves structures in the channel rather than passing around them. Specifically, the routine begins at the left edge of an "empty" plane, and proceeds to update the plane in a column by column sweep. If a net enters the plane at a particular column, the algorithm assigns the appropriate grid point to the entering net and if necessary rearranges the assignments of grid points in previous columns to maintain net continuity and minimize route width. If a net completely exits the plane, remaining nets "fall" into the empty space in order to guarantee that the final route will be of minimum width.

One difficulty with the basic plane router is that it places all nets as far towards the bottom of the channel as possible. This tends to generate nets with multiple-doglegs, as well as unnecessary wire runs between the top and bottom of the channel. To alleviate these problems, an optimizing routine is run on the plane after all nets in the B group have been routed. This routine straightens unnecessary doglegs, and moves nets as close as possible to their associated terminals. Figure 6-3 illustrates some of the operational features of the single layer router.

Nets not assigned to B groups are routed by an amortized version of YACR2 [83] that differs from that used in the original Chameleon only in that nets from other groups cannot initially be placed on a layer assigned to a B group. This restriction is a pragmatic one, based on a desire to avoid unnecessarily long computation times. In practice, the lack of an associated vertical routing layer virtually guarantees that a net not in a particular B group will be unroutable if placed on the group's associated layer. In all other ways, however, unused space on layers assigned to B groups is available for routing other nets.

While no experimental examples were in evidence, it is possible that the introduction of B layers will increase the required channel width, while reducing the number of required vias. A trade-off such as this is most likely to occur when the partitioner estimates that a partition using B layers will have area identical to a partition without B layers. In such cases, the inevitable reduction in available V layers makes the detailed routing more difficult, and thus favors the partition without B layers for minimum area. If area is not the critical parameter, however, the reliability advantages of fewer vias would speak in favor of the use of B layers. In practice, the IC designer should run MulCh with and without the use of B layers enabled, and choose the route that best matches the system design goals.

```
                After processing column 1:
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
    A   B   B   A               C   C   D       D       E
------------------------------------------------------------------
    A
    A
    A
    A
    A
    D
------------------------------------------------------------------
    D           F   F                       E   E
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
          Before adding/removing nets from column 4:
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
    A   B   B   A               C   C   D       D       E
------------------------------------------------------------------
    A   B   B
    A   B   B
    A   B   B
    A   B   B
    A   A   A   A
    D   D   D   D
------------------------------------------------------------------
    D           F   F                       E   E
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
          After adding/removing nets from column 4:
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
    A   B   B   A               C   C   D       D       E
------------------------------------------------------------------
    A   B   B   A
    A   B   B   A
    A   B   B   A
    A   A   A   A
    A   A   D   D
    D   D   D   F
------------------------------------------------------------------
    D           F   F                       E   E
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
                    All columns routed:
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
    A   B   B   A               C   C   D       D       E
------------------------------------------------------------------
    A   B   B   A               C   C   D       D       E
    A   B   B   A               C   C   D       D       E
    A   B   B   A               C   C   D       D       E
    A   A   A   A               C   C   D       D       E
    A   A   D   D   D   D       C   C   D   D   D       E
    D   D   D   F   F   D   D   D   D   D   E   E   E   E
------------------------------------------------------------------
    D           F   F                       E   E
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
                    After optimization:
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
    A   B   B   A               C   C   D       D       E
------------------------------------------------------------------
    A   B   B   A               C   C   D       D       E
    A   A   A   A                       D       D       E
    D   D   D   D   D   D   D   D   D   D   D   D       E
    D           F   F                       E   E   E   E
------------------------------------------------------------------
    D           F   F                       E   E
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Figure 6-3: Shown is a plane as it appears at various points during the routing of the indicated channel. The routing of column 4 is shown both before and after the nets for terminals at the column are added or removed. Note the displacement of nets D, A and B that is caused by the entrance of net F. In addition, note how the exit of net F in column 5 allows net D to “drop” into the bottom row of the plane. The final optimization removes redundant rows, unnecessary doglegs (nets A and D) and excess wire runs (net C).

6.5 Experimental Results

This section describes experimental results obtained with MulCh as compared with Chameleon.[4] Performance comparisons were made using a set of fourteen benchmark examples described below. The most dramatic improvements were seen in the four and seven-layer tests, where the mean reductions in channel width, total net length, and total number of vias were about 8%, 4%, and 20%, respectively.[5] MulCh routed several of the sample channels in width not only less than that of Chameleon, but less than the minimum theoretically possible without B layers.

Table 6.1 describes the example channels that were used to evaluate the performance of MulCh. Example channels include **3a**, **3b**, and **3c** from [109], the largest channel of the CMOS implementation of the SOAR microprocessor designed at Berkeley (**chris1**) [72], the contrived cyclic-VCG example (**cycle.t**) from [17], a revised version of the well-known Deutsch's difficult example [33] (**ddr**)[6], and four random problems (**r1**, **r2**, **r3**, and **r4**) generated by Rivest's program [85]. The examples **ex1** through **ex4** are difficult channels from an industrial standard cell chip provided by C. P. Hsu.

Table 6.2 lists the performance of MulCh and its improvements over Chameleon. The numbers enclosed in parentheses next to the channel width indicate the number of B layers used on each example. The columns labeled $\Delta\mathbf{R}$, $\Delta\mathbf{NL}$, and $\Delta\mathbf{V}$ give the differences between the number of rows, total net length, and total number of vias needed by MulCh and Chameleon to route a particular example. For convenience, entries under $\Delta\mathbf{NL}$ and $\Delta\mathbf{V}$ are listed as percentage reductions in Chameleon's results which were achieved by MulCh. Though all fourteen of the examples were tested on two through

[4] As stated earlier, MulCh was implemented by enhancing the Chameleon channel routing package. The version of Chameleon used here is actually a sightly revised version of the package described in [17]. Performance of the revised version is broadly equivalent to the earlier package, with isolated examples differing by at most a single row (generally in favor of the revised version). Since one purpose of the presented results is to demonstrate the inherent benefits of B layers, all comparisons are to the version of Chameleon that MulCh is based on.

[5] These means are computed by first expressing the results of MulCh for each channel as a percentage of the corresponding result for Chameleon. Then a geometric mean is computed and subtracted from 100%.

[6] This example is comprised of Deutsch's difficult example plus some extra nets filling in empty spaces.

Example	Density	Number of Columns	Number of Nets
3a	15	45	30
3b	17	62	47
3c	18	103	54
chris1	49	432	158
cycle.t	16	134	65
ddr	19	169	72
ex1	16	417	235
ex2	15	421	282
ex3	11	421	291
ex4	19	421	270
r1	20	139	77
r2	20	117	77
r3	16	123	78
r4	15	150	74

Table 6.1: Benchmark channels

seven layers, no B layers were used by MulCh in the cases omitted from the table. In these cases results obtained with MulCh are comparable, but not necessarily identical, to the results obtained by Chameleon. The previously mentioned simplifications to the partitioner cost function are responsible for the discrepancies.

The most interesting results are for four and seven layers, where the ability to easily increase the number of layers available for horizontal routing encourages the use of B layers. In the four-layer scenario, especially, MulCh often succeeded in simultaneously reducing the number of rows and vias, as well as total net length. Reductions in total net length and number of vias were also evident in cases where the number of rows was not reduced. The four-layer table is particularly relevant, since it represents technology which is currently becoming available. In the table for seven layers, fewer row number reductions appear than in the four-layer table; savings came mostly as reductions in total-net-length and the number of vias.

MulCh found only one opportunity for improvement when five and six layers were employed. For five and six layers, the number of layers available for horizontal routing does not increase with the addition of only a single B layer, and consequently the partitioner

Ex.	Width		Net Length		Vias	
4 Lyr.	New	ΔR	New	ΔNL	New	ΔV
3a	6(1)	2*	744	9.9%	53	24.3%
3b	7(1)	2*	1227	6.5%	85	17.5%
3c	8(1)	1*	1903	5.6%	126	15.4%
chris1	23(1)	2*	17128	1.3%	346	16.0%
cycle.t	8(1)	1	2676	1.1%	217	4.0%
ddr	10(1)	0	4047	0.9%	267	6.6%
ex1	8(1)	0	5342	3.1%	348	22.7%
ex2	7(1)	1*	5278	7.9%	349	31.7%
ex3	5(1)	1*	4363	7.5%	317	27.8%
ex4	9(1)	1*	5273	7.6%	316	29.0%
r1	10(1)	1	3285	4.0%	170	20.2%
r2	10(1)	0	2579	2.3%	124	23.5%
r3	8(1)	0	2248	2.6%	151	19.3%
r4	8(1)	0	2803	2.7%	199	18.4%
5 Lyr.	New	ΔR	New	ΔNL	New	ΔV
ex3	4(2)	0	4013	6.4%	192	56.3%
7 Lyr.	New	ΔR	New	ΔNL	New	ΔV
3a	4(1)	0	691	2.8%	57	16.2%
3b	4(1)	1*	1071	6.1%	83	19.4%
3c	5(1)	0	1670	1.8%	117	21.5%
chris1	12(1)	1*	15288	1.7%	345	16.3%
cycle.t	4(1)†	0	2203	3.8%	198	16.1%
ddr	5(1)	0	3324	3.1%	260	9.4%
ex1	4(1)†	0	4452	3.7%	321	28.3%
ex2	4(1)	0	4474	4.1%	337	32.2%
ex3	3(0)†	1	4030	5.0%	439	-0.5%
ex4	5(1)	0	4356	2.9%	338	24.7%
r1	5(1)	1	2858	3.3%	170	17.9%
r2	5(1)	0	2206	3.7%	128	21.0%
r3	4(1)†	1	1861	5.9%	150	20.2%
r4	4(1)	0	2431	3.8%	197	18.6%

Table 6.2: Results with MulCh

requires many coplanar nets to be successful.

Table 6.2 contains several entries worth noting. Of particular interest are the $\Delta\mathbf{R}$ entries marked with an asterisk, which indicate improvements over the optimal solutions obtainable without the use of B layers. Also of interest is example **ex3** for five layers, where a 56.3% reduction in vias was realized.

In only a few of the cases shown in the table (as indicated by a dagger next to the width), did MulCh achieve channel widths meeting the naive lower bound of density divided by the number of H and B layers. This is to be expected, since the planarity requirement greatly restricts the set of nets that can be placed in a B group.

While MulCh's heuristic-search nature makes a rigorous running time analysis impossible, it should be noted that all examples required less than eight minutes of μ**VAX** computer time. Additional code optimizations should result in further running time reductions.

Chapter 7

Faster Algorithms for Channel Routing Tasks

This chapter provides improved algorithms for certain tasks in channel routing, with special attention to time-consuming tasks performed by the multilayer channel router, MulCh described in Chapter 6. These improvements come along two main fronts. First, it is shown that certain single-layer routing problems can be solved in time linear in the number of nets. Of particular relevance to MulCh is the demonstration that the minimum width for a single-layer channel routing problem can be determined in linear time. Second, an incremental algorithm is given for computing channel density, so that each time a new net is added to a channel routing subproblem, the effect can be determined more quickly (logarithmic time) than by reprocessing the entire set of nets.

This chapter is divided into two sections. Section 7.1 discusses single-layer routing in channels and switchboxes, and Section 7.2 provides an incremental algorithm for computing density.

7.1 Single-Layer Channel and Switchbox Routing

This section provides a linear time algorithm for determining the minimum separation required to route a channel when the connections can be realized in one layer. It generalizes similar results for river routing by allowing single-sided connections. The approach can also be used to obtain a simplified routability test for single-layer switchboxes (routing in a rectangle). This section describes joint work with Miller Maley of Princeton U.

In recent years, a good deal of attention has been given to the problem of planar or single-layer wire routing for VLSI chips. Especially popular has been river routing in the restricted sense of the term, i.e., the connection of two parallel rows of corresponding points[1] [35, 67, 75, 90, 91, 97]. Other works have considered routing within a rectangle [24], placement and routing within a ring of pads [10], or routing between very general arrangements of modules [28, 66, 71].

These works have considered various wiring models (e.g. rectilinear or general) and several specific questions. Generally, there is some tradeoff between the generality of the wiring patterns considered and the sophistication of the questions asked. Asking for the best placement or routing of complicated arrangements of modules or wires would require solution of an NP-complete problem [51, 52, 55, 68, 78], but relatively sophisticated questions can be efficiently answered in the case of river routing. (See [75] for a particularly exhaustive list of specific problems in river routing.) In particular, the *minimum separation problem* [35, 67, 75, 90, 91] involves determining the minimum separation for two horizontal rows of terminals (in fixed horizontal positions) which will allow the routing to be performed. This problem can be solved in linear time under the rectilinear and most other wiring models [35, 67, 75, 90], less time than is required to perform the complete routing.

This section concentrates on showing that the minimum separation problem (in the rectilinear wiring model) can be solved in linear time for any single-layer channel routing

[1] This is the only usage of the term "river routing" in this thesis; more complicated variations of the problem are referred to as "single-layer" or "planar" routing.

problem, even with single-sided connections. In the process, it clearly provides a linear time solution to the *routability problem* which simply asks whether a routing is possible given a fixed vertical separation as well as fixed horizontal positions of the terminals. Later, extension to switchboxes will be discussed.

Figure 7-1 illustrates a single-layer channel routing problem and some of the notation to be used. The terminals along the top and bottom are labeled $t_1, t_2, \ldots, t_m$ and $b_1, b_2, \ldots, b_n$, respectively. In river routing, the problem would be to connect t_i to b_i for every i; in our more general channel routing problem we allow single-sided connections as between b_1 and b_4 in Figure 7-1.[2] For each each terminal T, we use the notation mate(T) to represent the terminal which is to be connected to T. We will also make the notation for terminals do double duty; where a terminal appears in an arithmetic context (comparison or subtraction), it will represent the x-coordinate or horizontal position of the terminal. It should be noted that restriction to two-point nets represents no loss of generality in the case of single-layer routing, since it is easy to transform problems with multiterminal nets to problems with only two terminal nets (but a larger number of nets) and to perform the reverse transformation on the routings obtained.

A few details remain in order to fully define the problem. Though wires are required to stay unit distance apart, it is convenient to assume that wires may be routed immediately alongside the channel boundaries. If practical considerations require that wires not run along the channel boundaries (except perhaps when connecting terminals on adjacent grid points along the top or bottom of the channel), the results in this section can be applied by simply routing wires one unit into the channel from each terminal, determining the minimum separation of the new rows of terminals, and adding two to obtain the separation of the original channel boundaries. The only case where this will not yield the optimal result is when all the nets are trivial in the sense that each net simply runs straight across the channel or connects adjacent grid points on the channel boundary. It

[2] We use the term "single-sided" for these connections even though the term "sides" is sometimes used to refer to the left and right of the channel as opposed to the top and bottom where the terminals lie.

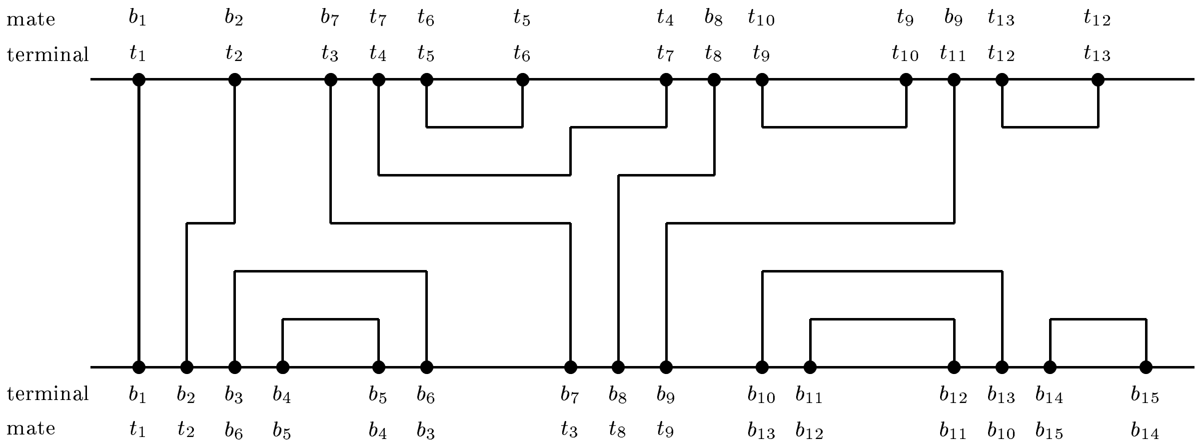

Figure 7-1: A representative single-layer channel routing problem. Each terminal T among the top terminals $\{t_i\}$ or the bottom terminals $\{b_i\}$ must connect to exactly one terminal $mate(T)$.

is easy to check for this special case, so this section assumes that routing along channel boundaries is permitted.

7.1.1 Finding Minimum Channel Separation

This section shows how to determine minimum separation for the channel routing problem just described. The first step in the demonstration is the invocation of a general theory of single-layer routing [28, 71] specialized to channel routing in order to obtain a set of necessary and sufficient conditions for routability. Then it is shown that we can reduce the number of conditions which must be checked, and, finally, that the minimum channel width satisfying the conditions can be determined in linear time.

It is most convenient for us to use the single-layer routing theory of [71] in a slightly altered form consistent with the approach of Cole and Siegel [28]. The basic idea of the theory is that routability can be decided by checking whether a limited collection of *cuts* are *safe*. Roughly, a cut is a line segment from top to bottom of the channel, and it is safe if there is enough room for all the nets which must cross it to do so. The alterations to Maley's theory were foreshadowed when it was indicated that we will allow routing along channel boundaries. It is convenient to have the terminals and the two channel boundaries play the role of *features* in the theory of [71]. Though this represents a change in that wires were required to stay unit distance away from nonterminal features in [71], it is only necessary to slightly alter a few definitions by considering cuts as closed rather than open line segments. The key definitions which place the theory in the form we desire are summarized below.

Definition: A *critical cut* is a line segment connecting a terminal on the top of the channel to a terminal on the bottom or a line segment running from a terminal straight across to the other channel boundary.

Definition: The *flow* of a cut c, denoted $\mathrm{flow}(c)$, is the number of wires which *must* cross c, including the wires incident at an endpoint of c.

Definition: The *capacity* of a cut c from b to t, denoted $\mathrm{cap}(c)$, is $\|b-t\|+1$, where $\|\cdot\|$ represents the ℓ_∞ norm, i.e., $\|(x,y)\| = \max\{|x|, |y|\}$.

Definition: A cut c is *safe* iff $\mathrm{flow}(c) \leq \mathrm{cap}(c)$.

We can now state formally, the result which we need from the general theory of single-layer routing.

Lemma 27 *A channel is routable if and only if all of the critical cuts are safe.*

Proof. The lemma follows immediately from the corresponding result in [71, p. 40] using slightly different definitions. ■

The restriction to checking critical cuts gives us $O(N^2)$ conditions which must be checked, where $N = m + n$ is the number of terminals. In the river routing problem, it can be shown that it suffices to check a smaller set of $O(N)$ cuts, but when single-sided connections are allowed it is not evident how to reduce the number of cuts below $\Omega(N^2)$. Nonetheless, it is possible to eliminate some of the critical cuts in a manner dependent on the flows and capacities of the cuts, leaving a set of cuts for which the flows and capacities are sufficiently nicely related that linear time suffices to determine the minimum channel width that makes all the cuts safe.

The necessary limitation on the critical cuts to be checked is provided by the simple distinction between *dense* and *sparse* cuts. A sparse cut is one which is safe regardless of the channel separation, i.e., the horizontal distance between the cut endpoints is large enough relative to the flow that the cut is guaranteed to be safe. Any cut which is not sparse is referred to as a dense cut. For convenience, we also classify as dense any cut running from a terminal straight across the channel. (Such a cut is both sparse and dense if its flow is 1.) Thus, any channel routing problem has dense cuts, and, in fact, any terminal is the endpoint of some dense cut. With the distinction between dense and sparse cuts, we can provide a simple expression for the channel separation as given in the following corollary to Lemma 27.

Corollary 28 *Minimum channel separation is given by*

$$-1 + \max_{c \text{ dense}} \text{flow}(c) \ .$$

■

The next key observation is that the set of dense cuts emanating from any given terminal form a "cone". In Figure 7-2, for example, the dotted and dashed lines show all the critical cuts emanating from terminal a on the bottom of the channel. Terminal l is the leftmost terminal on the top for which the cut from a is dense, and terminal r is the rightmost such terminal. The critical cuts to points between l and r inclusive are all dense, while the cuts outside of this cone are all sparse. To see that this situation prevails in general, just compare any sparse cut to a cut "immediately below" it. For example, in Figure 7-2, the sparsity of cut $[a, s]$ implies that cut $[a, s']$ must be sparse. This is true because the horizontal extent of $[a, s']$ is greater than that of $[a, s]$, but the flow of $[a, s']$ can be at most one greater than that of $[a, s]$. (In this example the flows are the same.)

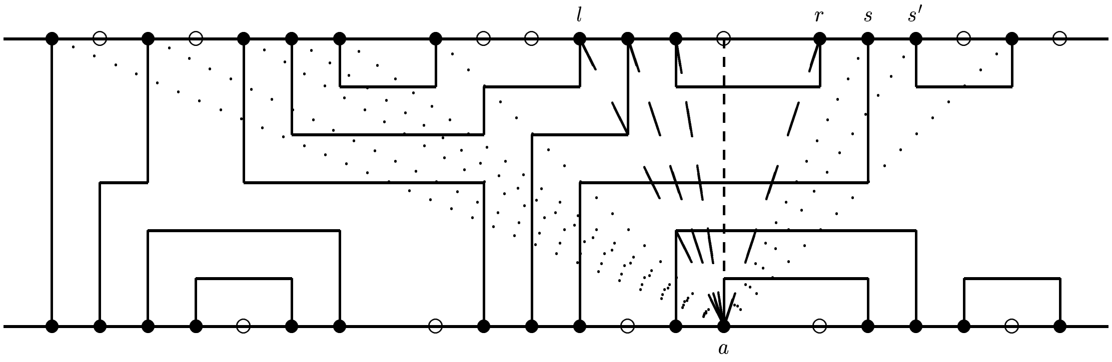

Figure 7-2: The dotted and dashed lines show the sparse and dense cuts, respectively, emanating from terminal a. In general, the dense cuts emanating from any terminal form a "cone". This figure also shows the dummy terminals (hollow circles) which are added for convenience in detailing the algorithm.

The basic idea of the algorithm for finding minimum channel separation is to sweep a cone of dense cuts across the channel, moving the apex of the cone along the bottom terminals. As we sweep across the channel, we compute the maximum of the flows of the dense cuts. By marching the apex of the cone only along the bottom, we may miss straight cuts emanating from top terminals, but this omission can be remedied by adding dummy bottom terminals at x-positions which have a top terminal but no bottom terminal. (For any dummy terminal T, mate(T) will have a special null value.) In fact, for simplicity of coding the algorithm, it is convenient to also add dummy top terminals at x-positions which have a bottom terminal but no top terminal. The positions of the dummy terminals are illustrated in Figure 7-2.

The cone sweeping approach is illustrated in Figure 7-3 which shows the cone of cuts as we move the apex from one bottom vertex to the next. In the figure, l and r are the leftmost and rightmost top terminals for which the cut from a is dense, and l' and r' play the same roles for the next bottom terminal a'. The main loop of the algorithm involves moving the apex of the cone through the bottom terminals from left to right. At each stage, we update the leftmost and rightmost top terminals corresponding to dense cuts, and, as will be explained further below, we maintain a collection of the flow values of cuts within the cone. The maximum of these flow values is compared to the maximum

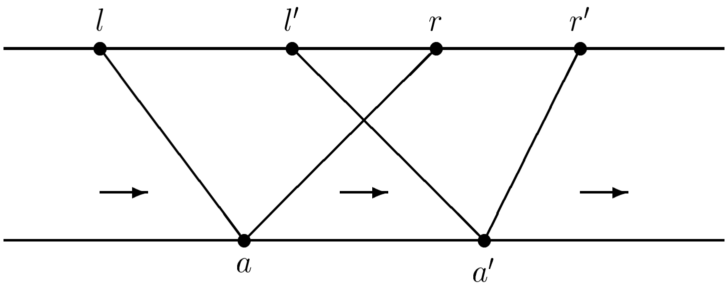

Figure 7-3: The dense cuts are checked by sweeping across the channel from one bottom to terminal to the next, maintaining a cone of dense cuts incident to the bottom terminal. At each stage, the algorithm keeps track of the maximum of the flows of all dense cuts seen so far.

seen so far so that the global maximum is known at the end of the sweep.

The first step in showing that the channel separation algorithm runs in linear time is to argue that each of the top boundaries of the cone of dense cuts moves monotonically across the channel. That is, with reference to Figure 7-3, l' is at least as far right as l, and r' is at least as far right as r. This a consequence of the fact that the dense cuts emanating from any terminal form a cone (as applied to a top terminal rather than a bottom terminal). This assures us that there are only a linear number of steps involved in moving the leftmost and rightmost top terminals of interest as the apex of the cone moves along the bottom terminals.

Now we simply need to show that we can maintain the maximum of the flows of the dense cuts without exceeding linear time as the cone sweeps across the channel.

First of all, note that each step of pushing the rightmost top terminal one terminal to the right is a constant time operation. It merely requires computing the flow of a new cut whose flow differs by -1, 0, or 1 from the flow of the "preceding" cut. Let us define a function flowdiff to represent this difference:

$$\mathrm{flowdiff}(p, q, r) = \mathrm{flow}([p, q]) - \mathrm{flow}([p, r]) \ .$$

When q and r are adjacent terminals on one of the channel boundaries, the value of $\mathrm{flowdiff}(p, q, r)$ can be computed by checking on which sides of the cuts $[p, q]$ and $[p, r]$

	flow	#cuts
	⋮	
	5	0
	4	0
max_flow →	3	2
	2	2
	1	1

flow_offset 3

Figure 7-4: The table used to tally the flows in the current collection of dense cuts.

the mates of q and r lie. As the rightmost top terminal is moved right, each new flow is associated to the relevant terminal and tallied in a table of flows in the current collection of cuts as illustrated in Figure 7-4.

The second operation we must treat is that of moving the leftmost top terminal one terminal to the right. This operation can be performed by looking up the already computed flow associated with the terminal being dropped from the cone at the left. Once we know this flow, we simply decrement the appropriate tally in the table of flows.

Finally, though it is less obvious, constant time also suffices to move the apex of the cone one terminal to the right while retaining the correct values of the flows of the cuts to the top terminals. This is true because when the apex of the cone moves one terminal to the right, the flows of cuts to relevant top terminals all change by the same amount (0, 1, or -1). That is, for all top terminals which are in the cone of dense cuts for both the old and the new apex, the change in flow is the same. A straightforward case analysis shows that if this were false, there would have been a cut of unit flow at the borderline between the cuts which experience different flow changes. But this cut and those below it would be sparse and therefore cannot be in the cone of dense cuts for both the old and new apex. Thus, when the apex of the cone moves, it is only necessary to adjust a flow offset which will be applied to all the flows in the table once the upper range of the cone is updated.

The only trick that remains is to keep track of the maximum flow of cuts in the table. Then after each movement of the apex and the appropriate adjustment of the leftmost and rightmost top terminals, this maximum flow plus the flow offset is compared to the running maximum over the whole channel. It is easy to update the maximum of flows in the table when a new flow is added; just compare the new flow to the old maximum. Fortunately, it is also unnecessary to do any searching when a flow is removed. Since adjacent cuts differ by flow at most one, the flow values which have nonzero tallies in the table will always be a continuous range of integers. Thus, to update the maximum flow in the table, it is only necessary to subtract 1 from the maximum if the flow being removed equals the old maximum and is the last flow of that value in the collection.

The arguments provided above suffice to show that the procedure in Figure 7-5 computes the minimum channel separation in linear time. Below is a more detailed guide to the code in Figure 7-5.

The variables in Figure 7-5 are used as follows. The variable *apex* represents the index among the bottom terminals of the apex of the cone of cuts. The variables *left* and *right* are used to keep track of the indices among the top terminals of the leftmost and rightmost endpoints of dense cuts emanating from the apex of the cone. The array *flow_to*, as modified by the variable *flow_offset* associates to top terminals delimited by *left* and *right* the flow of the cuts to these terminals from the apex of the cone. The array *num_cuts* corresponds to the table in Figure 7-4; for each possible value of *flow_to*, it keeps track of the number of top terminals delimited by *left* and *right* which correspond to that value. The variable *max_flow* is the maximum index of *num_cuts* for which the value of the array element is nonzero. That is, *max_flow* represents the maximum value of *flow_to* corresponding to a terminal delimited by *left* and *right*. Finally, *separation* is the running max (minus 1) of the flows of all dense cuts in the channel.

The initializations for procedure CHANNEL-SEPARATION are found in lines 1–6. Line 1 is used to conveniently ensure that straight cuts will be checked and the cone of dense cuts will never be empty. Initially, the apex of the cone is at the leftmost bottom terminal and the single cut in the cone is $[b_1, t_1]$. (The cone will be expanded later if necessary.) Lines 5–6 initialize the flow collection to contain just the information for the cut $[b_1, t_1]$.

```
procedure CHANNEL-SEPARATION
Add dummy terminals (with null mates) so that every x-position with a
   terminal has both a top and bottom terminal, yielding top and bottom
   terminal sets {t_1, t_2, ..., t_last} and {b_1, b_2, ..., b_last}.
apex ← 1
left ← 1 ; right ← 1
flow_offset ← 0 ; flow_to(1) ← flow([b_1, t_1])
      ⟨flow_offset + flow_to(right) = flow([b_apex, t_right]) always⟩
max_flow ← flow_to(1)
num_cuts(i) ← 0 for −n ≤ i ≤ ½(m + n) ; num_cuts(flow_to(1)) ← 1
while apex ≤ last
     while right < last
          flow_to(right + 1) ← flow_to(right) + flowdiff(b_apex, t_right+1, t_right)
          if t_right+1 − b_apex ≥ 1 ≥ flow_to(right + 1) + flow_offset − 1 then
               break        ⟨sparse cut⟩
          endif
          right ← right + 1
          num_cuts(flow_to(right)) ← num_cuts(flow_to(right)) + 1
          max_flow ← max {max_flow, flow_to(right)}
     endwhile
     while b_apex − t_left ≥ 1 ≥ flow_to(left) + flow_offset − 1        ⟨sparse cut⟩
          num_cuts(flow_to(left)) ← num_cuts(flow_to(left)) − 1
          if max_flow = flow_to(left) and num_cuts(flow_to(left)) = 0 then
               max_flow ← max_flow − 1
          endif
          left ← left + 1
     endwhile
     separation ← max {separation, max_flow + flow_offset − 1}
     apex ← apex + 1 and flow_offset ← flow_offset + flowdiff(t_right, b_apex, b_apex−1)
endwhile
```

Figure 7-5: This algorithm computes the maximum of the flows of the dense cuts, thereby determining the minimum channel separation for which all cuts are safe. Comments appear inside angle brackets.

Note that a range of $-n$ to $n+\frac{1}{2}(m+n)$ for the indices of *num_cuts* is certainly adequate since the flow of any cut is in the range $[1, \frac{1}{2}(m+n)]$, and the offset is in the range $[-n, n]$. Furthermore, for the cuts we consider at a given position of the apex, flow minus offset is upper bounded by $\frac{1}{2}(m+n)$. Hence the range of indices used in line 6. (Additional analysis may restrict the necessary range of indices further, but the range used is certainly sufficient.)

The main loop of procedure CHANNEL-SEPARATION, comprising the cone sweeping operation, is found in lines 7–26. There are four main parts to this loop. Lines 8–16 move the rightmost top terminal under consideration towards the right until the *next* cut from the apex would be sparse (and directed towards the right from the apex). Similarly, Lines 17–23 move the leftmost top terminal under consideration towards the right until the cut running from it to the apex is *not* sparse (or it is not to the left of the apex). The dense cuts then correspond to the top terminals which are at least as far right as *left* and at least as far left as *right*. Line 24 updates the running max of the flows of the dense cuts. Finally, line 25 moves the apex of the cone one terminal to the right along the bottom terminals and updates the offset to the table of flows being maintained. Within the loop that updates *right*, the main steps are computation of the new flow and updating of the flow collection and *max_flow* illustrated in Figure 7-4. Within the loop that updates *left*, the operations involved are to update the flow collection by looking up the flow value for the cut being dropped from consideration and then to update *flow_max* by taking advantage of the contiguous nature of the nonzero values in the flow collection as described above.

7.1.2 Extension to Switchboxes

This section considers extension of the above results from channels to switchboxes. That is, terminals are allowed at the sides of the channel instead of just at the top and bottom. With floating side-terminals, the problem is shown to easily reduce to the problem without side-terminals. When side-terminals are fixed, it is shown how to obtain a linear time routability test which is somewhat simpler than that of Chang and JáJá [24].[3]

The first variant of switch box routing, in which side-terminals are allowed to float to arbitrary y-positions may be thought of as merely an extended channel (figuratively and literally). (Included in this notion of floatability is that there are no single-sided side connections.) It suffices to extend the channel, place the side terminals along the top or bottom of the channel, and solve the problem as before. As long as there are no single-sided side connections, the wires attached to the former side terminals must cross the former channel sides, so any routing for the new problem can be converted to a routing for the old problem by merely truncating some wires.

When the side-terminals are fixed, a linear time routability test is obtained with little more than two applications of the procedure Channel-separation. After applying that procedure to the terminals at top and bottom of the switchbox and then applying it with the appropriate shift in orientation to the terminals at left and right of the switchbox, it is only necessary to check for safety of the cuts which connect perpendicular sides of the switchbox. Fortunately, checking these corner cuts is as easy as for river routing without side connections [28]; we need only check the diagonal cuts, which is a simple, linear time procedure.

7.2 Incremental Channel Routing

This section provides an algorithm to maintain channel density in logarithmic time per net insertion. Attention is restricted to the static case. That is, given a predetermined set of at most N columns known to contain all the terminals of the nets to be considered, it is shown that each time a net is added into the group under consideration, the density can be determined in $O(\lg N)$ time. The dynamic situation, in which the columns of interest are not known in advance, should be manageable by applying tree-balancing techniques from the literature to the tree-based algorithm to be described.

[3] Pinter also considers this problem [78, 79], but his linear time routability test contains a bug.

The algorithm for the incremental density computation works by maintaining data in the nodes of a complete binary tree T with leaves representing the N columns (in linear order). (For simplicity, N is assumed to be a power of 2.) Since the tree is complete, we can actually perform all our operations on a dense array rather than using pointers, but it is most convenient to visualize the algorithm in the tree setting. In order to describe the data to be maintained at nodes of T it is necessary to define the notion of a net covering nodes within the tree:

> **Definition:** A net *covers* any leaf node of T which corresponds to a column between the leftmost and rightmost terminal (inclusive) of the net. An internal node v of T is covered if all leaves in the subtree rooted at v are covered.

With respect to this definition of covering, two data items are maintained in every node of the tree T. The first is $d_{below}(v)$, which represents, roughly, the maximum density within the subtree rooted at v. More precisely, it is the maximum over all columns in the subtree of the density at that column, where attention is restricted to nets which do not cover the parent of v. The second data item kept in each node is $d_{at}(v)$, which is the density *at* v in contrast to (at or) *below* v. That is, it is the density of nets which cover v but not the parent of v, which is to say simply the number of nets covering v but not its parent.[4]

Maintaining d_{below} and d_{at} at the nodes of T suffices to maintain the overall density d, because we have

$$d = d_{below}(root) \ .$$

All that remains is to show that $\lg N$ time suffices to update d_{at} and d_{below} wherever necessary, each time that a new net is added to those under consideration. Note first

[4]The maintenance of d_{at} is just like the use of the segment tree data structure in several applications in computational geometry [81]. Routine operation on segment trees is sufficient to query for the density (in logarithmic time) at any point in a channel, but extra operations are needed to keep track of the maximum density.

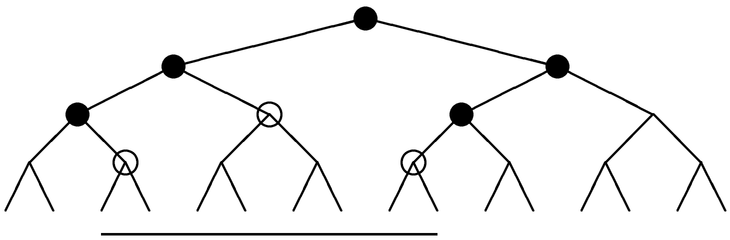

Figure 7-6: This figure shows the tree on the columns of interest, the span of a particular net, and the nodes at which data must be updated when the net is included. The circled nodes are the nodes where d_{at} and d_{below} must be incremented, and the nodes marked by solid disks are the additional nodes where d_{below} must be updated. There are at most $2 \lg N$ of each of these types of nodes.

that d_{at} merely needs to be increased by 1 at no more than $2 \lg N$ nodes. The relevant nodes are the roots of the maximal complete subtrees covered by the new net. Since there are at most two such subtrees of any given height (a consequence of the fact that any net covers a contiguous set of leaves), the number of nodes to be updated does not exceed $2 \lg N$. (It is straightforward to actually find and update the correct nodes.) It is also necessary to update d_{below} at no more than $4 \lg N$ nodes. These nodes are simply those where d_{at} was updated along with those appearing on the path to the root from the leftmost or rightmost node where d_{at} was updated. The form of the update at node v is just

$$d_{below}(v) \leftarrow \max\{d_{below}(\mathrm{left}(v)), d_{below}(\mathrm{right}(v))\} + d_{at}(v) \ ,$$

where left(v) and right(v) represent the left and right sons of v. Figure 7-6 shows an example of a span of columns covered by a net and the nodes where data must be updated.

Part III

Area Lower Bounds

Chapter 8

Lower Bounds on the Area of Finite-State Machines

There are certain straightforward algorithms for laying out finite-state machines. This chapter shows that these algorithms are optimal in the worst case for machines with fixed alphabets. That is, for any s and k, there is a deterministic finite-state machine with s states and k symbols such that *any* layout algorithm requires $\Omega(ks \lg s)$ area to lay out its realization. Similarly, any layout algorithm requires $\Omega(ks^2)$ area in the worst case for nondeterministic finite-state machines with s states and k symbols. In fact, we can say that "almost all" machines of s states and k symbols require nearly the specified area. This chapter is derived from a joint paper with Mike Foster of Columbia U. [40].

To manage the complexity of designing large VLSI chips, automated layout algorithms, sometimes called silicon compilers, are required. These are computer programs that accept a high-level behavioral description of a chip and produce its layout. There are many layouts corresponding to any behavioral specification, some better than others according to various measures. For example, some layouts may be very compact so that the chip can be small, some layouts may be very fast, so that the chip can work quickly, or some may be very low-power. We would like to be sure that our layout algorithms produce good layouts according to the measure we select.

Regular languages and finite automata have proven to be useful models for a wide range of computations. Many chips and portions of chips are commonly modeled as finite automata, and several systems have been built for automatic layout of these machines. Given a state-table or algorithmic description of a finite-state machine, these systems attempt to produce a small, fast layout on a VLSI chip. Techniques for layout have included regular expression trees [38, 39], networks of PLA's [98], and multilevel standard-cell logic [34].

This chapter shows that straightforward techniques for layout of regular languages are optimal in the worst case. Although some of the techniques that are used in practice can produce small layouts for some examples, there are many finite-state machines for which the straightforward implementation (such as direct coding of the state table) is the best possible. In particular, we show that for any s and k, any algorithm for laying out a finite-state machine must use area $\Omega(ks \lg s)$ for some deterministic machine with s states and k symbols. Asymptotically, this is the area of the direct state table implementation. Similarly, we show that $\Omega(ks^2)$ area is required for some nondeterministic machine with s states and k symbols, which corresponds to a straightforward implementation of nondeterministic machines.

This chapter is organized to highlight the ideas behind the lower bounds. We prove a simple theorem that displays the method and then extend it to cover more complicated cases. In Section 8.1 we describe the model of computation in which these lower bounds hold. Section 8.2 proves the lower bound on the area of a fixed-shape region for layout of deterministic finite-state machines with 2 character alphabets, and Section 8.3 proves a similar lower bound for nondeterministic machines. Section 8.4 extends the results to machines with larger alphabets. In Section 8.5 we show that allowing different machines to occupy regions of different shapes does not decrease the area required.

A few more bibliographic notes are in order. A key step in obtaining the lower bounds in this chapter is to obtain a lower bound on the number of functionally distinct deterministic and nondeterministic finite-state machines. A bound for deterministic machines, similar to the bound derived here by elementary means, was independently obtained by Harrison [47] as a corollary to the derivation of formulas that can be used to calculate the exact numbers of "nonisomorphic" deterministic finite-state machines. Similar formulas are developed by Harary and Palmer [46] using slightly differing definitions of nonisomorphic finite-state machines.

8.1 Area Lower Bounds in VLSI

Why prove only a lower bound on area? Several types of lower bounds on the resources needed by VLSI chips have been proven [18, 21, 92, 96]. Most of these have been bounds on the product of chip area and some power of the running time; AT^2 is common. In this chapter we prove a lower bound for area alone, without considering time. Finite-state computations differ from the computations considered by previous authors (context-free language recognition, sorting, DFT, etc.); the area required by a computation on a string of length n is independent of n. Computations on which bounds have previously been proven (including an area lower bound on string matching [92]) have required increasing area as the length of the input increases. The memory required by a finite-state machine is fixed, no matter how long the input may be. Moreover, the time taken by a finite-state computation is simply the time to read the input. As the length of the input increases, then, the computation time increases proportionally while the area stays constant, so that the tightest AT^k lower bound on a finite-state computation on an input of size n is simply $\Omega(n^k)$. Therefore, we can characterize the implementation of a finite-state machine strictly in terms of its area.

To prove a lower bound on area, we need a model of computation that relates area to the problem to be solved. We will use a model most similar to that of Thompson [96] as discussed in Section 1.1. Any computational structure must be laid out on a grid of unit squares, corresponding to the minimum feature size of the implementation technology. Each of the unit squares may contain one or more types of conducting or insulating material in several layers. There is a finite number of possible conducting or insulating layers, and a unit square may contain some combination of these layers. The combination chosen for a particular unit square can be specified by a fixed number of bits, so the set of combinations chosen for an area A can be specified by a number of bits proportional to A. The set of layers chosen for each square of the chip determines its structure completely, which determines the computation performed. We have therefore proven the following fundamental lemma.

Lemma 29 *The computation performed by an area of size A can be specified with $\Theta(A)$ bits.*

In a typical CMOS implementation, for example, the unit squares are typically about one micron on a side, and each square may contain p or n-type diffusion, polysilicon, and one or two layers of metal. There may also be insulators present to isolate these layers, or the insulators may be removed to make a contact. There are about 60 combinations of conducting layers and insulators that make electrical sense, so the configuration of each unit square can be specified by 6 bits. Therefore, the computation performed within an area of m square microns can be specified by at most $6m$ bits. Of course, many of the possible combinations will not perform meaningful computations, since neighboring squares might violate design rules, or the layout as a whole might not be a working circuit. The constant of proportionality is therefore less than 6, but it is clear that the computation performed by a CMOS chip of area m can be specified by $\Theta(m)$ bits.

To prove a lower bound of area $\Omega(n)$ using this model, we show that the computation we want to perform requires n bits of specification. Suppose that we are given a chip of area A, together with a family of 2^n different computations, any one of which may be specified as the circuit to be built on the chip. Since the computations are all different, each one requires a different structure for area A. The structure of area A is thus specified by n bits, so A must be $\Omega(n)$.

In using this proof technique, notice that the computations themselves must be different, not just the computational structures. If two different structures on the same chip produced the same behavior, an implementor could always use just one of them in implementing a computation. By counting behaviors rather than structures, we can be sure that we need a different chip for each different computation.

The results we obtain from this technique are worst-case bounds. They show that at least one member of the family of 2^n computations will require area n. The best case could of course be considerably better. For example, we show below that the worst-case area for laying out an s-state deterministic machine is $\Omega(s \lg s)$. Many s-state machines have considerably smaller area, requiring as little as $O(\lg s)$ area for the best layout. The bounds in this chapter, while existentially tight, do not eliminate the search for clever implementations.

8.2 Deterministic Automata

There is a well-known method for simulating a deterministic finite automaton (DFA) with a program for a general-purpose computer. States of the machine are represented by integers, which are used as indices into a transition table. Position i of the table contains the next states for all input characters if the machine is in state i. If the program is in state i, it selects its next state from position i in the table, and uses that state to index into the table on the next character input.

This program structure can be implemented in hardware in a straightforward fashion using a PLA. The PLA for a machine with s states and 2 symbols in the alphabet will have $\lg s$ input and output columns to represent the states as well as an input line for the alphabet symbol. The number of rows (product terms) in the PLA will be $2s$, one for each combination of state and input symbol. The output state bits are computed by ORing the appropriate product terms into each state bit. The total area of this hardware realization of the finite-state machine is $O(s \lg s)$.

Of course, for many machines the PLA implementation is extremely wasteful of space. For example, an n-bit counter has 2^n states so that its PLA representation would require area $n \times 2^n$. A simple ripple-carry implementation of the same counter, however, requires only area n. A clever layout algorithm for finite-state machines might be able to recognize that a particular machine was a counter and use the smaller implementation. There are other tricks of this kind for other types of machines, such as machines that recognize patterns having short regular expressions [38, 39, 98]. Do all finite-state machines succumb to these kinds of techniques, or are there machines for which the obvious PLA layout is the best?

In the worst case, the PLA layout is the best that can be done. We can show this by producing a set of s^s different languages that can each be accepted by machines with s states. It is not sufficient to simply note that there are s^{2s} different state tables for s-state machines, since some of these state tables represent equivalent machines. A clever layout algorithm could implement equivalent machines in the same way. We must count languages or behaviors, not machines. Keeping this in mind, we can prove the following theorem, which provides a lower bound matching the upper bound for area of an s state DFA.

> **Theorem 30** *The amount of area required to implement an s state deterministic finite automaton is $\Omega(s \lg s)$ in the worst case.*

Proof. Consider the following class, F_s, of s state machines with input alphabet $\{0, 1\}$. Each machine has states numbered 0 to $s-1$, where state 0 can be distinguished from all others by observing the output of the machine; it can be thought of as a final state if the machine is used as a recognizer. The machines act identically in every state on input 0; specifically, if a machine is in state i and gets an input of 0, it goes to state $i+1 \bmod s$. On input 1, however, all possible behaviors are included among the machines in F_s. Since the transition on input 1 from each of the s states can be to any one of s states, there are s^s machines in F_s.

The machines in F_s represent different languages, so they can be distinguished by their behaviors. To see this, consider two different machines in F_s. There must be some state i for which a 1 input takes the machines to different states, say j and k. We can test for the presence of an edge from i to j by inputting zeros to put the machine into its distinguished state, then inputting the string 0^i10^{s-j}. The machine will be in its distinguished state at the end of this input if and only if there is a transition from state i to state j on input 1. Thus if two machines in F_s differ, there is a string accepted by one but not by the other.

The machines in F_s all behave differently, so any layout algorithm must produce different layouts for each one. Since there are s^s machines in this set, $s \lg s$ bits are required to specify one of the layouts. Therefore, an area A that can contain the layout of any of the machines in F_s must be of size $\Omega(s \lg s)$. ■

8.3 Nondeterministic Automata

Floyd and Ullman [38] presented a generic layout for a non-deterministic finite automata (NFA) that is analogous to the layout given above for deterministic machines. The only difference is that instead of encoding the state into $\lg s$ PLA inputs and outputs, one bit is used for each of the s states. After each input symbol is received, the PLA output corresponding to a given state is on if and only if the non-deterministic machine could be in that state. The area of the Floyd-Ullman implementation of an s-state NFA over a two-symbol alphabet is $O(s^2)$ since there are s input and output terms and at most $2s$ product terms.

Is there a better way to implement non-deterministic machines? In a proof similar to that for the deterministic case, we can show that $\Omega(s^2)$ is a lower bound on the layout area.

> **Theorem 31** *The amount of area required to implement an s state nondeterministic finite automaton is $\Omega(s^2)$ in the worst case.*

Proof. Once again, consider a set F_s of machines over the two-symbol alphabet $\{0, 1\}$. The states of each machine are numbered 0 to $s - 1$, with state 0 distinguishable by the output of the machine. The set F_s consists of those machines such that the only 0-edge out of state i goes to state $i + 1 \bmod s$. Every possible combination of 1-edges occurs in some machine in F_s. Since the transitions on input 1 from each state can be to any subset of the s states, F_s contains $(2^s)^s$, or 2^{s^2} machines.

The machines in F_s all recognize different languages, by the same argument used in the deterministic case. If two machines differ, there must be a 1-edge present in one machine that is not present in the other one. Thus we can distinguish different machines by testing for the presence of a single 1-edge. As above, to determine whether the 1-edge from state i to state j is present in a machine, we place the machine in state 0, then input the string 0^i10^{s-j}. Since the machines in F_s are deterministic on 0 inputs, the machine will be in state 0 after this input if and only if there is a 1-edge from i to j.

Since the machines in F_s represent different languages, they must all have different layouts. The specification of a non-deterministic machine with s states is therefore a specification of one of 2^{s^2} different layouts. By the arguments given in the preceding sections, the smallest chip that can contain any of these layouts has area $\Omega(s^2)$. ■

8.4 More Than Two Symbols

If a machine has more than two symbols in its alphabet, both the upper and lower bounds for layout area increase. An s-state deterministic machine with k symbols in the alphabet has a PLA implementation with area $O(ks \lg s)$. To see this when k is $O(s)$, we use a construction just like the earlier one, except that there are $\lg s + \lg k$ PLA inputs in order to account for both the states and input symbols, and there are ks product terms. If k is not $O(s)$, then we can use an alternative construction which yields an area bound of $O((k + \lg s)s \lg s)$, which is $O(ks \lg s)$ for $k = \Omega(\lg s)$.

The alternative upper-bound construction is obtained by using a PLA in which the input symbol has been decoded, i.e., it is represented by turning on exactly one of k input lines. (It is actually desirable to input the complements of these signals to the PLA.) Each product term corresponds to the pairing of a state and an output state bit which is turned on by some transitions from that state. The product term includes the check that the input symbol is not one which fails to generate such a transition. Since there are

$s \lg s$ product terms and $k + \lg s$ input terms, the area of the PLA is $O((k + \lg s)s \lg s)$.

To decode input symbols specified with $\lg k$ bits into the k-bit representation, we just need to add the area of a decoder, which is $k \lg k$. The decoder is certainly small enough if $\lg k$ is $O(s \lg s)$. If $\lg k$ is greater than $O(s \lg s)$, then some of the input symbols are extraneous. That is, there are only s^s complete state-transition behaviors which can be assigned to the input symbols, so if there is a larger number of input symbols, there must be symbols with identical behaviors. Thus it suffices to translate the $\lg k$ input signals into $s \lg s$ signals corresponding to the appropriate state-transition behavior in binary (requiring area $s \lg s \lg k$), and then decode these $s \lg s$ signals into the appropriate s^s signals in unary (requiring area $s^s s \lg s$, which is $O(ks \lg s)$ for $\lg k = \Omega(s \lg s)$).

The lower bound on deterministic machines with s states and k symbols is also $\Omega(ks \lg s)$, as we now show.

Theorem 32 *The amount of area required to implement an s state, k symbol DFA is $\Omega(ks \lg s)$ in the worst case.*

Proof. Consider the family F_s, extended so that each state has k outgoing edges instead of just two. For each of the s states, one of the outgoing edges (the 0-edge) is constrained, but each of the other $k - 1$ edges can go to any one of s next states. Each state therefore has s^{k-1} possibilities, and since all states are independent the number of possible machines is $(s^{k-1})^s$. The machines are all different, since the presence of any edge can be tested by experiments on the machine. The number of bits required to specify one of these machines, and thus the layout area required, is therefore $\Omega(ks \lg s)$. ∎

The extension to k symbols is similar in the case of nondeterministic finite automata. The area of an s state NFA with k symbols is $O(ks^2)$. The straightforward modification of Floyd and Ullman's construction gives a PLA with $s + \lg k$ input terms and ks product terms for an area of $O((s + \lg k)ks)$, which is $O(ks^2)$ when s is $\Omega(\lg k)$. If s is not $\Omega(\lg k)$, we can use an alternative construction which yields an area bound of $O((k + s)s^2)$, which is $O(ks^2)$ when s is $O(k)$.

The alternative upper bound is obtained in the same fashion as described above for deterministic machines. When decoded alphabet symbols are input to the PLA, the number of inputs plus outputs is $O(k+s)$, and the number of product terms is s^2, for an area of $O((k+s)s^2)$. Again we can handle input symbols which have not been decoded with a decoder of area $k \lg k$. The area of the decoder is $O(ks^2)$ if s is $\Omega(\sqrt{\lg k})$, and otherwise we have redundant symbols. That is, there are only 2^{s^2} possible state-transition behaviors for each symbol. Thus the $\lg k$ input bits for the alphabet symbols can be translated into s^2 bits representing the state-transition behavior and then decoded into 2^{s^2} bits, all in area $s^2 \lg k + s^2 2^{s^2}$, which is $O(ks^2)$ for $s = O(\sqrt{\lg k})$.

As before, we obtain a lower bound which matches the upper bound on area.

Theorem 33 *The amount of area required to implement an s state, k symbol NFA is $\Omega(ks^2)$ in the worst case.*

Proof. We once again consider the family of machines F_s, in which 0-edges form a cycle. There are $(2^s)^{s(k-1)}$ machines in F_s since from each state, there are 2^s possible combinations of edges for each character. A similar proof to those above shows that these machines all recognize different languages, and thus must have different layouts, so the area lower bound for a non-deterministic machine with s states and a k-symbol alphabet is $\Omega(s^2 k)$. ■

8.5 Shape Independence

The results which we have given so far hold in the case of laying out machines in a fixed region such as a square. The results depend only on the mild assumption that there is a minimum size grid, with each grid square having a finite number of possible compositions.

In fact, stronger assumptions are reasonable for VLSI technologies, and we can show that all the previous results hold even if we allow each machine to be implemented in a different area of arbitrary shape. In VLSI technologies, adjacent grid squares containing conducting material on the same layer are electrically connected, and there is no other relevance to the shape in which the grid squares are arranged as long as the design rules are obeyed. It is therefore possible to view a VLSI layout as a bounded-degree graph of grid squares of various compositions.

We extend our earlier results by invoking the following lemma, which states that $\Theta(A)$ bits suffice to specify all graphs of wire area A (i.e. A grid squares).

Lemma 34 *There are $2^{O(A)}$ VLSI layouts of wire area A.*

Proof. We begin by invoking a result of Thompson that a graph of wire area A must have a bisection width of $O(\sqrt{A})$ [96]. Then we can write a recurrence for $G(A)$, the number of graphs of wire area A as follows:

$$G(A) \leq [G(A/2)]^2[(A/2)^2]^{O(\sqrt{A})} \ .$$

The right side of the recurrence is an upper bound on the number of ways of forming two graphs of wire area at most $A/2$ and then forming connections across the bisection. Letting $H(A) = \lg G(A)$, we can rewrite the recurrence as

$$H(A) \leq 2H(A/2) + O(\sqrt{A} \lg A) \ .$$

Then noting, that $G(1)$ is a constant, we can solve the recurrence to obtain $H(A) = O(A)$, and $G(A) = 2^{O(A)}$. ■

8.6 Conclusion

This chapter has presented results on area lower bounds for finite-state automata. We have shown that any algorithm for laying out finite automata must require as much area for some machines as do the most straightforward algorithms. Our results can be extended easily to volume lower bounds in 3D-VLSI circuits, as long as there is a minimum feature size.

Can our bounds be tightened? Our lower bounds are tight for only the worst case machines with a given number of states. However, there are finite-state machines with s states that can be laid out in considerably less than $s \lg s$ area. For example, an s-state counter can be laid out in area $\lg s$. What features of finite-state machines let them have small layout area? It would be desirable to obtain a simple measure of complexity of finite-state machines in terms of which we could obtain universally close upper and lower bounds on area.

Even if tighter bounds are found for some types of finite-state machines, it is clear that almost all machines will require the area that our bounds specify. For example, consider the set of s-state deterministic machines that have layouts of area s. There are at most 2^s different machines that can be laid out in area s, so the fraction of s-state machines that can have area s layouts is at most $2^s/s^s$, which is asymptotically zero. Thus, almost all deterministic machines with s states require more than area s. Similar arguments show that, for any area lower bound $L(s)$ in this chapter and for any function $f(s)$ such that

$$\lim_{s \to \infty} \frac{2^{f(s)}}{2^{L(s)}} = 0$$

almost all machines with s states require more than $f(s)$ area.

Tighter bounds than ours must depend upon machine features which are present in a vanishingly small proportion of s-state machines. Nonetheless, machines of practical interest may contain these features, so a search for tighter bounds is worthwhile.

Chapter 9

Conclusion

This thesis has made contributions in three areas: the design of general-purpose parallel computers, multilayer channel routing for VLSI chips, and implementation of finite-state machines. This section summarizes the key findings and suggests further areas of investigation.

The discussion of general-purpose parallel computers can be summarized simply: it is feasible to construct good general-purpose parallel machines. Specifically, an appropriate universal machine of a given hardware cost (measured as area or volume) incurs only a small polylogarithmic penalty in simulating any other machine of comparable cost.

One of the most important open questions about universal networks has already been mentioned in Chapter 4: Can we do any better than slowdown bounded by the square of the logarithm of network size? At present, no nontrivial lower bounds for the worst-case simulation time are known.

Additional unresolved issues relate to the practical implementation and use of fat-tree based machines. In practice, it is undesirable to *always* pay a polylogarithmic or even constant overhead for the privilege of using a general-purpose machine. The simulation results in this thesis provide confidence that the universal networks presented are robust and general-purpose, but we would like to be able to obtain greater efficiency by programming the networks directly, and we would like to be able to execute architecture-independent programs in a way that will usually yield better results than the worst-case simulation overhead. Progress in these directions has been made by Leiserson and Maggs [65], but much work remains to be done. Also, the use of asymptotic analysis in this thesis leaves open the choice of some constant factors which must be specified in any actual machine implementation. Finally, packaging issues are deserving of investigation; it would be desirable to build large universal networks using a modest number of different chips as in Bhatt and Leiserson's packaging scheme for ordinary trees [14].

The main claim of this thesis regarding multilayer channel routing is that existing tools can benefit from the incorporation of layers which are not dedicated to a single routing direction. Application of this approach to a leading tool has yielded significant improvements on example problems in terms of channel width, total net length, and total number of vias.

This thesis has also shown how to efficiently perform important subtasks in multilayer channel routing, specifically determination of the minimum required channel width for a single-layer problem and incremental computation of density as nets are brought into consideration one by one. Still open, however, are incremental computation of minimum channel width for single-layer problems and incremental computation of longest path lengths in vertical constraint graphs.

An additional useful area of investigation would be to compare the channel router presented in this paper to the routing algorithm of Berger, Brady, Brown, and Leighton [12], which has a "provably good" worst-case performance. Their algorithm applied to an L-layer problem of density d will produce a routing of width $\frac{d}{L-2} + O(\sqrt{d/L} + 1)$, which could be quite attractive depending on the exact size of the additive term and the ability of the algorithm to do better than the worst case on practical problems. It would be desirable to create a practically oriented implementation of this algorithm and to ascertain its performance on example problems.

The last part of this thesis showed that straightforward and standard approaches to implementing finite-state machines are optimal in the worst case. As mentioned earlier, however, special classes of finite-state machines may require much less area than the worst-case bound. Many researchers have worked on the problem of reducing the area required by finite-state machines, but we still have little insight as to what types of finite-state machines may significantly benefit from these efforts.

Bibliography

Following each reference, in brackets, is a list of pages where that reference is cited.

[1] Harold Abelson and Peter Andreae. Information transfer and area-time tradeoffs for VLSI multiplication. *Communications of the ACM*, 20–23, January 1980. ⟨21⟩

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974. ⟨64⟩

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 1–9, ACM Press, 1983. ⟨22, 33⟩

[4] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983. ⟨22, 33⟩

[5] R. Aleliunas. Randomized parallel communication. In *Proceedings of the 1st ACM Symposium on Principles of Distributed Computing*, pages 60–72, ACM Press, 1982. ⟨22, 31⟩

[6] Romas Aleliunas. *Probabilistic Parallel Communication.* PhD thesis, Department of Computer Science, University of Toronto, 1982. TR no. 166/83. ⟨22⟩

[7] Helmut Alt, Torben Hagerup, Kurt Melhorn, and Franco P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, October 1987. ⟨22⟩

[8] B. Awerbuch, A. Israeli, and Y. Shiloach. Efficient simulation of PRAM by ultracomputer. 1983. ⟨22⟩

[9] Brenda S. Baker, Sandeep N. Bhatt, and Tom Leighton. An approximation algorithm for manhattan routing. In Franco P. Preparata, editor, *VLSI Theory.* Volume 2 of*Advances in Computing Research*, pages 205–229, JAI Press, 1984. ⟨93⟩

[10] Brenda S. Baker and Ron Y. Pinter. An algorithm for the optimal placement and routing of a circuit within a ring of pads. In *24th Annual Symposium on Foundations of Computer Science*, pages 360–370, IEEE Computer Society Press, 1983. ⟨110⟩

[11] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. *A General Method for Solving Divide-and-Conquer Recurrences.* Technical Report CMU-CS-78-154, Department of Computer Science, Carnegie-Mellon University, December 1978. ⟨64⟩

[12] Bonnie Berger, Martin Brady, Donna Brown, and Tom Leighton. Nearly optimal algorithms and bounds for multilayer channel routing. Unpublished manuscript. ⟨95, 140⟩

[13] Sandeep N. Bhatt and Frank Thomson Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, April 1984. ⟨12, 27, 58, 63, 65, 68, 72, 77, 79–80, 84–85⟩

[14] Sandeep N. Bhatt and Charles E. Leiserson. How to assemble tree machines. In Franco P. Preparata, editor, *VLSI Theory.* Volume 2 of*Advances in Computing Research*, pages 95–114, JAI Press, 1984. ⟨140⟩

[15] G. Bilardi, M. Pracchi, and F. P. Preparata. A critique and appraisal of VLSI models of computation. In H. T. Kung, Bob Sproull, and Guy Steele, editors, *VLSI Systems and Computations*, pages 81–88, Computer Science Press, 1981. ⟨12⟩

[16] Martin L. Brady and Donna J. Brown. Optimal multilayer channel routing with overlap. In Charles E. Leiserson, editor, *Advanced Research in VLSI; Proceedings of the Fourth MIT Conference*, pages 281–296, MIT Press, April 1986. To appear in *Algorithmica.* ⟨95⟩

[17] Douglas Braun, Jeffrey L. Burns, Fabio Romeo, Alberto Sangiovanni-Vincentelli, Kartikeya Mayaram, Srinivas Devadas, and Hi-Keung Tony Ma. Techniques for multi-layer channel routing. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, 7(6):698–712, June 1988. ⟨16, 91, 95, 105⟩

[18] R. P. Brent and L. M. Goldschlager. Area-time tradeoffs for VLSI circuits. In *Microelectronics '82*, pages 52–56, The Institution of Engineers, Australia, May 1982. ⟨129⟩

[19] R. P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *Journal of the ACM*, 28(3):521–534, July 1981. ⟨12, 21⟩

[20] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM*, 25(4):581–595, October 1978. ⟨64⟩

[21] R. P. Brent and H. T. Kung. On the area of binary tree layouts. *Information Processing Letters*, 11(1):46–48, August 1980. ⟨129⟩

[22] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, C-31(3):260–264, March 1982. ⟨21⟩

[23] Peter Bruell and Paul Sun. A "greedy" three layer channel router. In *IEEE International Conference on Computer-Aided Design (ICCAD-85)*, pages 298–300, IEEE Computer Society Press, 1985. ⟨94⟩

[24] Shing-Chong Chang and Joseph JáJá. Optimal parallel algorithms for river routing. 1988. Unpublished manuscript. ⟨110, 121⟩

[25] Bernard Chazelle and Louis Monier. A model of computation for VLSI with related complexity results. *Journal of the ACM*, 32(3):573–588, July 1985. ⟨13⟩

[26] Yun Kang Chen and Mei Lun Liu. Three-layer channel routing. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-3(2):156–163, April 1984. ⟨94⟩

[27] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952. ⟨38⟩

[28] Richard Cole and Alan Siegel. River routing every which way, but loose. In *25th Annual Symposium on Foundations of Computer Science*, pages 65–73, IEEE Computer Society Press, 1984. ⟨110, 112, 121⟩

[29] Jingsheng Cong, D. F. Wong, and C. L. Liu. A new approach to three- or four-layer channel routing. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, 7(10):1094–1104, October 1988. ⟨94⟩

[30] Thomas H. Cormen and Charles E. Leiserson. *A Hyperconcentrator Switch for Routing Bit-Serial Messages.* Technical Report MIT/LCS/TM-321, Laboratory for Computer Science, Massachusetts Institute of Technology, February 1987. Earlier version in *Proceedings of the 1986 International Conference on Parallel Processing.* ⟨33⟩

[31] Ido Dagan, Martin Charles Golumbic, and Ron Yair Pinter. Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, 21:35–46, 1988. ⟨97⟩

[32] Alvin M. Despain and David A. Patterson. X-Tree: a tree structured multiprocessor computer architecture. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 144–151, ACM/IEEE, 1978. ⟨66⟩

[33] D. Deutsch. A 'dogleg' channel router. In *Proceedings of the 13th ACM/IEEE Design Automation Conference*, pages 425–433, IEEE Computer Society Press, 1976. ⟨105⟩

[34] S. Devidas, H-K. Ma, A. R. Newton, and A. Sangiovani-Vincentelli. Mustang: state assignment of finite-state machines for optimal multi-level logic implementation. In *ICCAD-87*, pages 16–19, IEEE, November 1987. ⟨128⟩

[35] Danny Dolev, Kevin Karplus, Alan Siegel, Alex Strong, and Jeffrey D. Ullman. Optimal wiring between rectangles. In *Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 312–317, ACM Press, 1981. ⟨110⟩

[36] R. J. Enbody and H. C. Du. Near-optimal n-layer channel routing. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 708–714, IEEE Computer Society Press, 1986. ⟨95⟩

[37] W. Feller. *An Introduction to Probability Theory and Its Applications.* Volume 1, John Wiley, New York, 2 edition, 1957. ⟨38⟩

[38] R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *JACM*, 29(3):603–622, July 1982. ⟨128, 132–133⟩

[39] M. J. Foster. A specialized silicon compiler and programmable chip for language recognition. In N. G. Einspruch, editor, *VLSI Design*, chapter 5, pages 139–196, Academic Press, Orlando, Florida, 1986. ⟨128, 132⟩

[40] M. J. Foster and Ronald I. Greenberg. Lower bounds on the area of finite-state machines. *Information Processing Letters*, 30(1):1–7, January 1989. ⟨127⟩

[41] Zvi Galil and Wolfgang J. Paul. An efficient general-purpose parallel computer. *Journal of the ACM*, 30(2):360–387, April 1983. ⟨22⟩

[42] Ronald I. Greenberg, Alexander T. Ishii, and Alberto L. Sangiovanni-Vincentelli. MulCh: A multi-layer channel router using one, two, and three layer partitions. In *IEEE International Conference on Computer-Aided Design (ICCAD-88)*, pages 88–91, IEEE Computer Society Press, 1988. ⟨92⟩

[43] Ronald I. Greenberg and Charles E. Leiserson. A compact layout for the three-dimensional tree of meshes. *Applied Mathematics Letters*, 1(2):171–176, 1988. ⟨77⟩

[44] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.* Volume 5 of*Advances in Computing Research*, JAI Press, 1989. To appear. Earlier versions available in MIT/LCS/TM-307 and *26th Annual Symposium on Foundations of Computer Science*, 1985, pages 241–249. ⟨24, 30, 54, 58–59, 77, 85⟩

[45] Susanne E. Hambrusch. Channel routing algorithms for overlap models. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-4(1):23–30, January 1985. ⟨95⟩

[46] Frank Harary and Ed Palmer. Enumeration of finite automata. *Information and Control*, 10:499–508, 1967. ⟨129⟩

[47] Michael A. Harrison. A census of finite automata. *Canadian Journal of Mathematics*, 17:100–113, 1965. ⟨129⟩

[48] Akihiro Hashimoto and James Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th ACM/IEEE Design Automation Conference*, pages 155–169, IEEE Computer Society Press, 1971. ⟨94⟩

[49] W. Heyns. The 1-2-3 routing algorithm or the single channel 2-step router on 3 interconnection layers. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 113–120, IEEE Computer Society Press, 1982. ⟨94⟩

[50] Anna R. Karlin and Eli Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 160–168, ACM Press, 1986. ⟨22⟩

[51] Gershon Kedem and Hiroyuki Watanabe. Graph-optimization techniques for IC layout and compaction. In *Proceedings of the 20th ACM/IEEE Design Automation Conference*, pages 113–120, IEEE Computer Society Press, 1983. ⟨110⟩

[52] Mark R. Kramer and Jan van Leeuwen. *Wire-Routing is NP-Complete.* Technical Report RUU-CS-82-4, University of Utrecht, the Netherlands, February 1982. ⟨110⟩

[53] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980. ⟨33⟩

[54] Andrea S. LaPaugh and Ron Y. Pinter. *Channel Routing for Integrated Circuits (a survey).* Technical Report YALEU/DCS/TR-713, Yale University Department of Computer Science, June 1989. To appear in the *Annual Review of Computer Science*, Vol. 4, 1989. ⟨94⟩

[55] Andrea Suzanne LaPaugh. *Algorithms for Integrated Circuit Layout: An Analytic Approach.* PhD thesis, Department of Electrical Engineering & Computer Science, Massachusetts Institute of Technology, November 1980. MIT/LCS/TR-248. ⟨94, 110⟩

[56] F. T. Leighton. *Complexity Issues in VLSI.* MIT Press, Cambridge, Massachusetts, 1983. ⟨23, 77, 79⟩

[57] F. T. Leighton and A. L. Rosenberg. Three-dimensional circuit layouts. *SIAM Journal on Computing*, 15(3):793–813, August 1986. ⟨15, 77–79, 81, 84–85⟩

[58] F. Tom Leighton. Parallel computation using meshes of trees. In M. Nagl and J. Perl, editors, *Proceedings 1983 Workshop on Graphtheoretic Concepts in Computer Science*, pages 200–218, Trauner Verlag, 1983. ⟨77, 85⟩

[59] Frank Thomson Leighton. New lower bound techniques for VLSI. *Mathematical Systems Theory*, 17:47–70, 1984. ⟨77, 79⟩

[60] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the 16th ACM Symposium on Theory of Computing*, pages 71–80, ACM Press, 1984. ⟨22⟩

[61] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–269, IEEE Computer Society Press, 1988. ⟨35, 52–54, 56, 60, 69⟩

[62] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, C-34(10):892–901, October 1985. ⟨13, 23, 27, 31–32, 54, 58, 72, 77, 84–86⟩

[63] Charles E. Leiserson. Area-efficient graph layouts (for VLSI). In *21st Annual Symposium on Foundations of Computer Science*, pages 270–281, IEEE Computer Society Press, 1980. ⟨64, 77⟩

[64] Charles E. Leiserson. VLSI theory and parallel supercomputing. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 5–16, MIT Press, 1989. ⟨24⟩

[65] Charles E. Leiserson and Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988. ⟨26, 140⟩

[66] Charles E. Leiserson and F. Miller Maley. Algorithms for routing and testing routability of planar VLSI layouts. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 69–78, ACM Press, 1985. ⟨110⟩

[67] Charles E. Leiserson and Ron Y. Pinter. Optimal placement for river routing. *SIAM Journal on Computing*, 12(3):447–462, August 1983. ⟨110⟩

[68] T. Lengauer. On the solution of inequality systems relevant to IC-layout. *Journal of Algorithms*, 5(3):408–421, September 1984. ⟨110⟩

[69] Bruce M. Maggs. Personal communication. ⟨69⟩

[70] F. Miller Maley. Personal communication, October 1984. ⟨46⟩

[71] F. Miller Maley. *Single-Layer Wire Routing*. PhD thesis, Department of Electrical Engineering & Computer Science, Massachusetts Institute of Technology, August 1987. MIT/LCS/TR-403. ⟨110, 112–113⟩

[72] C. Marino. *Smalltalk on a RISC - CMOS Implementation*. MS Report, Department of EECS, University of California, Berkeley, 1985. ⟨105⟩

[73] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, MA, 80. ⟨11, 13⟩

[74] Kurt Melhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984. ⟨22⟩

[75] Andranik Mirzaian. River routing in VLSI. *Journal of Computer and System Sciences*, 34:43–54, 1987. ⟨110⟩

[76] James K. Park. Personal communication, 1989. ⟨35, 52–53, 56⟩

[77] M. S. Paterson. *Improved Sorting Networks with $O(\log N)$ Depth*. Research report 89, Department of Computer Science, University of Warwick, January 1987. ⟨22⟩

[78] Ron Y. Pinter. River routing: methodology and analysis. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 141–163, Computer Science Press, March 1983. ⟨110, 121⟩

[79] Ron Yair Pinter. *The Impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits*. PhD thesis, Department of Electrical Engineering & Computer Science, Massachusetts Institute of Technology, August 1982. MIT/LCS/TR-291. ⟨97, 121⟩

[80] N. Pippenger. Parallel communication with limited buffers. In *25th Annual Symposium on Foundations of Computer Science*, pages 127–136, IEEE Computer Society Press, 1984. ⟨22, 31⟩

[81] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction. Texts and Monographs in Computer Science*, Springer-Verlag, New York, 1985. ⟨122⟩

[82] Abhiram G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, IEEE Computer Society Press, 1987. ⟨22, 53⟩

[83] James Reed, Alberto Sangiovanni-Vincentelli, and Mauro Santomauro. A new symbolic channel router: YACR2. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-4(3):208–219, July 1985. ⟨103⟩

[84] John H. Reif and Leslie G. Valiant. “A logarithmic time sort for linear size networks”. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 10–16, ACM Press, 1983. ⟨22⟩

[85] R. L. Rivest. “Benchmark” channel-routing problems. Unpublished manuscript, 1982. ⟨105⟩

[86] Ronald L. Rivest and Charles M. Fiduccia. A "greedy" channel router. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 418–424, IEEE Computer Society Press, 1982. ⟨94⟩

[87] John E. Savage. Area-time tradeoffs for matrix multiplication and related problems in VLSI models. *Journal of Computer and System Sciences*, 22:230–242, December 1981. ⟨21⟩

[88] Charles L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7, Addison-Wesley, Reading, MA, 1980. ⟨13⟩

[89] C. H. Séquin, A. M. Despain, and D. A. Patterson. Communication in X-TREE, a modular multiprocessor system. In *ACM 78: Proceedings 1978 Annual Conference*, pages 194–203, 1978. ⟨66⟩

[90] Alan Siegel and Danny Dolev. The separation for general single-layer wiring barriers. In H. T. Kung, Bob Sproull, and Guy Steele, editors, *VLSI Systems and Computations*, pages 143–152, Computer Science Press, 1981. ⟨110⟩

[91] Alan Siegel and Danny Dolev. Some geometry for general river routing. *SIAM Journal on Computing*, 17(3):583–605, June 1988. ⟨110⟩

[92] O. Sykora and I. Vrto. Tight chip area lower bounds for string matching. *Information Processing Letters*, 26(3):117–119, November 1987. ⟨129⟩

[93] Thomas G. Szymanski. Dogleg channel routing is NP-complete. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-4(1):31–41, January 1985. ⟨94, 100⟩

[94] Steven L. Tanimoto. *Towards Hierarchical Cellular Logic: Design Considerations for Pyramid Machines.* Technical Report 81-02-01, Department of Computer Science, University of Washington, February 1981. ⟨66⟩

[95] C. D. Thompson. Area-time complexity for VLSI. In *Proceedings of the 11th ACM Symposium on Theory of Computing*, pages 81–88, ACM Press, 1979. ⟨11–12, 21–22⟩

[96] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1980. ⟨11–13, 21–22, 65, 79–80, 129, 137⟩

[97] Martin Tompa. An optimal solution to a wire-routing problem. *Journal of Computer and System Sciences*, 23:127–150, 1981. ⟨110⟩

[98] H. W. Trickey. Good layouts for pattern recognizers. *IEEE Transactions on Computers*, C-31(6):514–520, June 1982. ⟨128, 132⟩

[99] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, July 1984. ⟨22, 31⟩

[100] Eli Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 241–250, ACM Press, 1989. ⟨22⟩

[101] Eli Upfal. A probabilistic relation between desirable and feasible models of parallel computation. In *Proceedings of the 16th ACM Symposium on Theory of Computing*, pages 258–265, ACM Press, 1984. ⟨22⟩

[102] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. In *25th Annual Symposium on Foundations of Computer Science*, pages 171–180, IEEE Computer Society Press, 1984. ⟨22⟩

[103] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982. ⟨22, 31, 38⟩

[104] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 263–277, ACM Press, 1981. ⟨22, 31⟩

[105] Leslie G. Valiant. Universal circuits. In *Proceedings of the 8th ACM Symposium on Theory of Computing*, pages 196–203, ACM Press, 1976. ⟨21⟩

[106] Leslie G. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Computers*, C-30(2):135–140, 1981. ⟨77⟩

[107] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983. ⟨22⟩

[108] Jean Vuillemin. A combinatorial limit to the computing power of V.L.S.I. circuits. In *21st Annual Symposium on Foundations of Computer Science*, pages 294–300, IEEE Computer Society Press, 1980. ⟨21⟩

[109] Takeshi Yoshimura and Ernest S. Kuh. Efficient algorithms for channel routing. *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-1(1):25–35, January 1982. ⟨94, 105⟩