THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

A CLASS OF CELLULAR COMPUTER ARCHITECTURES TO SUPPORT PHYSICAL DESIGN AUTOMATION

Robin Arthur Rutenbar

CRL-TR-35-84

September 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

ABSTRACT

A CLASS OF CELLULAR COMPUTER ARCHITECTURES TO SUPPORT PHYSICAL DESIGN AUTOMATION

by

Robin Arthur Rutenbar

Chairman: Daniel E. Atkins

Special computer architectures for design automation (DA) problems are a practical solution to manage the increasing complexity of designing large integrated systems. A class of cellular architectures is studied which is applicable to physical DA problems that are cellular in nature: problems well-represented on a cellular grid with strongly local functional dependencies. The Raster Pipeline Subarray (RPS) class is a systolic organization, the central features of which evolved historically in classical picture-processing applications. RPS-structured DA engines have several attractive engineering properties: transparent expansion of processing capacity with a linear pipeline, direct accommodation of large grids with the raster data format, and application to differing tasks with programmable pipeline stages.

This thesis studies routing and integrated circuit design rule checking in an RPS environment. An experimental hardware/software RPS environment is constructed around existing RPS hardware; tools are developed to support the design and debugging of large-scale RPS-based DA systems. Maze-routing is the major application of interest; a mapping of maze-routing algorithms onto an RPS pipeline is developed and its complexity analyzed. A progression of increasingly complex, fully functional routers is implemented in the prototype RPS environment to verify feasibility. Large-scale benchmarks and comparisons with software routers show significant speedups.

Design rule checking is considered from an algorithmic viewpoint: it is shown that a formalism derived from picture-processing tasks provides an elegant conceptual and notational tool for mapping rules checking onto an RPS pipeline. Hardware implementations of some specifications are analyzed. These studies prove that existing RPS engines can be improved incrementally and gracefully: additional pipeline stages improve execution times. Experiments over a range of pipeline lengths, and extrapolations based on experimental measurements support this claim. From these experiments, detailed performance models and cost/performance metrics are developed, treated analytically, and rigorously optimized. These results are employed to deduce the necessary functionality and tradeoffs to design optimal RPS-structured DA engines.

A CLASS OF CELLULAR COMPUTER ARCHITECTURES TO SUPPORT PHYSICAL DESIGN AUTOMATION

by

Robin Arthur Rutenbar

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer, Information and Control Engineering)
in The University of Michigan
1984

Doctoral Committee:

Professor Daniel E. Atkins, Chairman
Professor John P. Hayes
Professor Ronald J. Lomax
Associate Professor Trevor N. Mudge
Professor Kensall D. Wise

TABLE OF CONTENTS

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

LIST OF APPENDICES

APPENDIX

CHAPTER I

INTRODUCTION

1.1. Problem Statement

The successful implementation of increasingly complex integrated systems demands the existence of increasingly sophisticated design automation tools. Traditional DA research--for example, mathematical analysis of DA algorithms and data-structures, application of software structuring techniques to mask layout, and use of databases to manage the design process--has produced software tools running on conventional serial computers. These tools are limited in three fundamental ways: by the inherent complexity of the problem, by the efficiency of the coded implementation, and by the resources of the machine on which the code runs. To overcome these three limitations recent attention has focused on special-purpose hardware for DA tasks. The strategy is to structure a machine architecture to exploit the task's inherent parallelism, replace software with hardware and firmware, and include precisely those resources critical to the task's solution.

The increasing cost of solving such DA problems can render general-purpose serial computers incapable of supporting a particular DA task. This application-dependent expense, measured in execution time or necessary machine resources, may arise because a problem is inherently too large, or because the preferred solution method requires complex or costly algorithms unaffordable when applied to a realistic problem. For example, a simple design rule check on the layout of a small mask is straightforward, but the same check performed on an entire microprocessor layout may be so expensive as to preclude performing the check as often as desired; the difficulty arises from the problem size. Similarly, maze-routers offer a wide

range of sophisticated performance, but may be unaffordable in practice due to the costly cell-by-cell processing of the grids representing the routing task; the difficulty here arises because the basic method is computationally demanding. Moreover, as technology advances permit the construction of more sophisticated systems, there is no guarantee that currently tractable design tools will accommodate these advanced designs, even on the enhanced general-purpose machines that will be built with such technologies. These concerns motivate research about special DA architectures.

This thesis examines a class of cellular architectures suitable for problems in physical design automation with respect to these concerns. A class of architectures called *raster pipeline subarrays* is examined; individual members of the class are called *RPS* machines. An RPS machine is a pipeline of subarray stages which processes a large grid as a serial, raster-order cell-stream. Machines in this class are appropriate for physical design tasks that are well represented on a fixed cellular grid and characterized by local functional dependencies among cell neighborhoods. In this framework, a large problem is one that must be represented on a large grid, and a complex solution method is one that requires computationally intensive processing of single grid cells or local cell groups. Tasks such as placement, routing, design rule checking, and device extraction have previously been solved by employing a grid representation and primarily local processing. This thesis concentrates on the design and analysis of routers in an RPS environment, and also examines briefly IC mask operations such as design rule checks in this environment.

Two central features of the RPS class address the difficulties that prohibit general-purpose machines from executing large DA tasks at an acceptable cost:

- **Serial Cell-Stream**

  Because RPS architectures process a grid in raster order--that is, as a serial stream of grid cells--the entire grid need never reside on an RPS machine. Thus, if extremely large grids can be generated they can be processed directly as they stream through an RPS-structured DA engine.

- **Pipeline Structure**

  Parallelism in the form of a pipeline enables RPS machines to perform several concurrent operations on grid cells streaming through the pipeline; it is this parallelism that enables RPS machines to be *fast* DA engines. Programmable pipeline stages provide the flexibility to accommodate a range of complex DA tasks adequately modeled in a cellular framework. Modular pipeline stages provide a practical mechanism to achieve precisely the performance required for the tasks at hand: small machines with short pipes can be upgraded to more powerful machines by the addition of pipeline stages.

Direct accommodation of a range of problem sizes and applications, and incremental expansion of processing power are the unique features that distinguish the RPS class as particularly cost-effective for DA tasks. This thesis examines in detail how these features affect the formal design, implementation, and analysis of RPS-structured DA engines.

1.2. Research Overview

The research undertaken in this thesis can be partitioned into four broad, related areas:

- **The Evolution of RPS Architectures as DA Architectures**

As part of the overall goal to show how RPS-structured machines can function as DA engines, we survey and critique related architectures and applications. New machine architectures often appear as evolutions or generalizations of previous structures motivated by new applications. This is the case with the RPS class: in this thesis the RPS class is proposed as the appropriate abstraction of some of the novel features introduced by machines such as the *cytocomputers* in the field of cellular image processing [LoMS80, LoMc80]. Related cellular processors are reviewed and a taxonomy of these processors is constructed. This effort serves (1) to show how similar cellular problems have been approached with different machine structures, (2) to motivate the RPS organization, and (3) to delineate its relationship to and evolution from other cellular organizations. Similarly, DA architecture research is reviewed and critiqued in order (1) to show the range of DA problems being tackled, (2) to study how similar problems are handled on different DA machines, and (3) to motivate the typical metrics used to

evaluate DA hardware. Our cellular taxonomy is employed to categorize DA architectures intended for grid-based problems (those problems for which RPS machines are appropriate); several connections are made between such grid-based DA machines and general cellular architectures. The intent of this work to characterize the niche occupied by RPS-structured DA engines in the spectrum of DA hardware. We suggest the RPS structure as a new *paradigm* for DA machines, in precisely the sense that the standard parallel array has been the abstract model for several DA engines.

- **Experimental Studies of DA Tasks on RPS Machines**

An existing family of RPS machines, the cytocomputers, is employed for experimental implementation of several DA tasks.[1] The experiments range from initial feasibility studies to mature, nearly complete systems. A new software environment is designed to support the development of these DA algorithms for this RPS hardware. Concrete implementations and performance statistics are examined for routing systems and DRC sub-systems functioning on the hardware. A principle goal of the experimental work is to identify performance bottlenecks, separating and abstracting those generic to RPS systems from those endemic to the current hardware.

- **Performance Evaluation and Optimization**

The study of RPS architectures partitions naturally into *local* issues and *global* issues. Local issues involve the processing of individual grid cells or cell groups as a grid streams through a pipeline of subarray processors. Local concerns pertain primarily to the functional design of pipeline subarray stages and the algorithms that execute within these stages. Global issues concern strategies for handling complete grids in a pipeline environment; the design of such strategies must account for system parameters such as pipeline bandwidth and length, and host overhead. Given a specific task such as a routing problem, a *solution* for a specific RPS hardware environment comprises the encoding of the problem as a cellular grid, the design of algorithms to execute in the RPS pipeline stages and the design of algorithms to move the

[1]By our definition, this family is a subset of the RPS class whose pipeline subarray stages are intended for picture-processing. These machines should not be regarded as DA engines because they are not *designed* to accommodate DA tasks. Despite a sub-optimal architecture for DA problems, these machines provide a useful research vehicle to realize basic variants of important DA tasks such as maze-routing and DRC.

necessary pieces of the grid through the pipeline. In this framework we examine how to achieve optimal performance for specific problems and solution techniques. Typically, solutions with minimum execution time are desired. It is shown that achieving such optimality mandates an accurate model of the basic performance of a solution method in an RPS environment. Such a model is parameterized by the values of the global constants for a specific machine realization. These models allow us to determine the best solution for a specific RPS realization. Moreover, they allow us to predict a good solution for our experimental systems, which is verified by measurement.

- **Architectures for RPS-Structured DA Engines**

Given a specific DA task and a set of performance goals, for example, routing a particular class of layouts within fixed time and cost constraints, we study how to design an appropriate RPS-structured DA engine that meets these goals. Local issues are briefly addressed here because it is essential to match the functional capabilities of each RPS stage to the set of problems to be solved. RPS stage architectures with different performance characteristics are suggested for routing and IC mask operations; these proposed structures are based primarily on experience with experimental systems. Although local subarray stage functionality does impact the optimal global strategy for data movement, we concentrate on global design. Several stage architectures embodying radically different tradeoffs and, moreover, incurring radically different costs may *each* suffice to satisfy a set of performance constraints. We argue that it is naive to embark on a detailed stage design without knowledge of how to evaluate abstractly the alternatives and choose the best from among them. Hence, global design is addressed by employing the models previously developed to optimize the performance of specific algorithms. One goal is to evaluate the sensitivity of these algorithms to variations in global parameters, e.g., is a long, slow pipe the same as a short, fast pipe? Systematic variation of these parameters generates a set of candidate RPS systems; solving for the best performance achievable with each candidate effectively partitions the design space into classes of machines with equivalent performance. This analysis is used to evaluate the effects of cost/performance tradeoffs for different hardware configurations.

1.3. Thesis Organization

The subsequent chapters of this thesis motivate our definition of the RPS class and focus on the solution of routing and DRC problems on RPS-structured DA engines.

Chapter II discusses the evolution of RPS architectures and provides the background necessary to evaluate their application as DA engines. Related cellular architectures are surveyed and a compact taxonomy is constructed to categorize them. DA architectures are likewise reviewed and critiqued, as are the metrics used to evaluate DA machines. Parallels are drawn between these cellular machines and DA engines intended for grid-based DA problems; the cellular taxonomy is employed to clarify the relationships among grid-based DA engines.

Chapter III defines and characterizes the RPS class. Important antecedents to the RPS organization, including cellular algorithms and machine implementations, are summarized. It is argued that the RPS class is the appropriate abstraction of the novel features introduced by these antecedent machines. Divorced from implementation details and application constraints, this definition is the appropriate level from which to proceed toward new areas such as DA engines. A design methodology for RPS structured systems is then suggested. The engineering tradeoffs implicit in the class are examined and compared with parallel array structures. The experimental hardware/software environment developed to support our RPS DA experiments is presented.

Chapter IV examines maze-routers in an RPS environment. A progression of increasingly complex routers implemented in our prototype RPS environment is presented. We discuss the essential local and global design issues for each router. These include the algorithms that execute in pipeline stages to mark cells in the routing grid, and algorithms to move the appropriate sections of the entire routing grid through the pipeline with the greatest economy. These experimental systems illustrate fundamental performance issues common to *any* RPS system. Analytical models are constructed to address these issues in the general case and optimal solutions are obtained. Solutions obtained by varying model parameters enable us to extrapolate the system configuration most suitable to meet any specific performance constraints. Further, these models are used to accurately predict the measured performance of empirically tuned experi-

mental systems.

Chapter V considers design rule checking in an RPS environment. In contrast to previous research on grid-based DRC, this work concentrates on the algorithmic specification of the individual checks comprised by a complete DRC. It is shown that a formalism originally developed for binary image analysis, suitably applied, provides an elegant and precise tool for the specification of DRCs. An experimental implementation of an example specification is constructed and its performance analyzed. Other IC mask operations are briefly addressed.

Chapter VI investigates the design of RPS-structured engines intended specifically to support DA tasks. The fundamental restrictions and bottlenecks discovered in the experimental systems are reviewed briefly. Alternative RPS stage architectures embodying different tradeoffs in functionality are suggested. We concentrate on global issues such as the design of system-level parameters which must be adequately characterized before informed decisions can be made about detailed stage architectures. The sensitivity of performance metrics is examined with respect to variations in global system parameters. The models developed in earlier chapters are employed to generate candidate RPS systems embodying different hardware tradeoffs to satisfy a set of inflexible performance constraints.

Chapter VII presents concluding remarks, summarizes the basic contributions of this research, and suggests some areas for further study related to the RPS organization.

CHAPTER II

BACKGROUND: CELLULAR ARCHITECTURES AND DA ENGINES

2.1. Introduction

This chapter connects research in the fields of cellular architecture--RPS architecture specifically--and DA architecture. A central theme of this thesis is that the RPS class represents a new approach to grid-based DA problems. To support this assertion, we survey related cellular architectures because the RPS structure is itself a cellular architecture. An extensive survey of DA engines is also presented, along with an analysis of the goals and metrics by which such machines are typically evaluated. Parallels are then drawn between architectures for cellular processing and for grid-based physical DA: each is characterized by how storage and processing power are allocated to cells in the problem. It is shown that several DA engines intended for grid-based applications can be well-regarded as instances of more general cellular structures, and hence, that architectures for general cellular problems are potential solutions for grid-based DA problems.

2.2. Cellular Architectures

It is important first to be specific about the notion of a *cellular* architecture. For our purposes, a cellular architecture is simply one which supports a cellular grid as the primary data structure. This definition includes special hardware designed specifically for grid-based tasks, and excludes machines such as conventional mainframes that, while certainly applicable, are not strictly intended for these tasks. Note also that machines composed primarily of replicated components, that is, of replicated *cells*, are not necessarily *cellular* machines by this definition,

unless they are applicable to grid-based tasks. Similarly, machines need not be composed of replicated elements (cells) to be suited to cellular problems.

Research in special-purpose hardware for cellular applications spans more than two decades and has accelerated with recent advances in technology. An immediate candidate for such problems is a cellular array, and indeed, advances in technology have heralded a renaissance for arrays in many applications, including DA. However, classical two-dimensional arrays are not the only machine organization capable of efficient solution of grid-based DA problems. Architectures for solving grid-based problems have been studied extensively in fields such as image processing, pattern recognition, and mesh-based numerical analysis.

This section focuses on machines that support image-processing and pattern-recognition [PDLN79, DaLe81, DuLe81, PrUh82, Kido83, Pres83]. Discussion is restricted to these machines for three reasons. First, much of the work on cellular architectures has been in the picture-processing area because of the precise match between problem and architecture; many of the cellular machines actually constructed are intended for picture-processing activities. Second, this thesis is concerned with those DA tasks that can loosely be regarded as *symbolic* processing, for example mask layout operations, which more resemble picture-processing applications like pattern-recognition than they resemble numerical-computation. Third, most of the antecedents to the RPS class arose from tasks related to picture-processing; it is from solutions to some of these concrete problems that we later abstract the central features of the RPS organization. From the narrow perspective of grid-based DA we construct a taxonomy of these cellular processors emphasizing:

- how storage is allocated to the cells of a grid being processed,
- how processing power is applied to individual cells or groups of cells,
- how processing elements and storage elements are interconnected.

The place of RPS architectures is described in this scheme. It will be shown later that many grid-based DA architectures fit naturally into this scheme.

A central problem for a cellular architecture that manipulates large grids is how to distribute all grid cells over a numerically smaller set of processors. In image-processor architecture

this problem has been referred to as *windowing* [Pres83]. Because the grids representing real images span the range 10^2 to 10^5 cells on a side, it is generally impossible to allocate a unique physical processor to each cell. Instead, the grid must be manipulated in subsections or windows. For our purposes, the shapes of these discrete sections, their acquisition, their path to and from processing elements, and the amount of parallelism in data-movement and data-manipulation define the architecture.

Fig. 2.1 shows a taxonomy emphasizing these features. It has three salient points. First, because the ultimate goal here is to illuminate DA architectures, this scheme is just large enough to contain most of the interesting grid-based DA architectures of which we are aware; cellular architectures that have no close analogue in current DA machines (e.g., pyramid machines) are simply omitted. Accordingly, it is not intended to be formal in the sense that the categories within the taxonomy are rigid, definitive partitions. Second, it is explicitly a

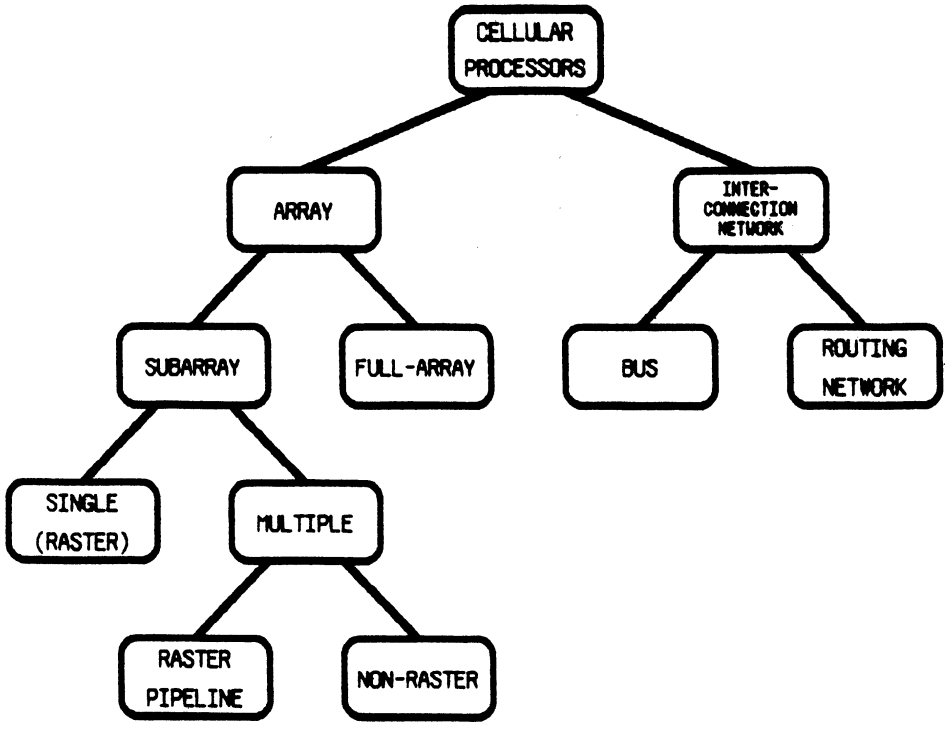


Figure 2.1 A Taxonomy of Cellular Processors

hierarchical classification in contrast to other schemes [DaLe81, Pres83]. Specifying a machine by its parents in the hierarchy gives its concise relationship to other machines, emphasizing critical similarities and differences. Third, it places RPS machines in this scheme to show their natural relationship with other array organizations.

At the first level the hierarchy divides into two basic machine organizations. As noted in [DaLe81] there are machines whose architectures are dominated by a central bus structure or, more generally, by an *Interconnection-Network (ICN)* structure. The other basic organization is, as expected, the *array* structure. By array structure we mean specifically the existence of one or more spatially distributed matrices of locally connected processor/storage elements and the machinery to move data through these elements. The critical distinction between ICN and array structures is that arrays are primarily *locally* connected--processing elements can be considered to be embedded in a grid or lattice--whereas ICN systems admit more general, global connectivity. Each of these two basic organizations is subdivided into two classes.

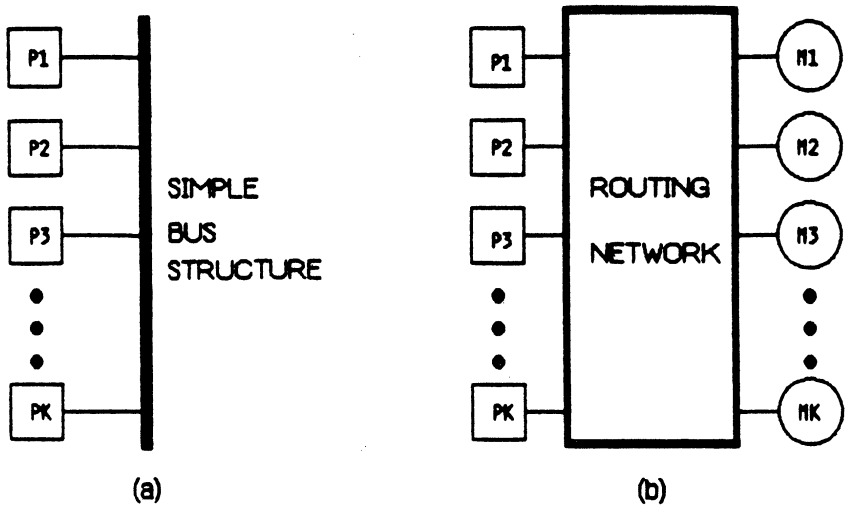


Figure 2.2 ICN Structure

(a) Bus-based structure. (b) Network-based structure.

ICN-structured machines are classified as using either a *bus* or a *routing-network*, as shown in Fig. 2.2. Although bus-based structures can be considered as a degenerate form of a network, we prefer to view them as a separate category. The label *bus-structured* is taken to imply systems which have few physical communication paths and in which most data transfers are multiplexed in time. Routing networks, on the other hand, imply more complex physical connectivity and more concurrent data transfers. For example, the PICAP II machine [Anto82] is centered around a single high-speed bus connecting image memories, a neighborhood processor, and a filter processor. In contrast, the proposed PASM architecture [Sieg81] employs multi-path routing-networks to connect a set of processor/memory subsystems.

The class of array-structured machines is also divided into two subclasses. Adopting the terminology of [Pres83], array-structured machines are classified as being *subarrays* or *full-arrays*. The distinction here requires clarification, as it depends not only on structural differences, but also on the size of the array involved. The full-array would be labeled a traditional cellular array: a matrix of processor/memory pairs each connected locally to its neighbors (Fig. 2.3). Early arrays include SOLOMON [SIBM62], and the Illiac machines [McCo63, Barn68]. Modern machines in this class are typically large square arrays of simple bit-

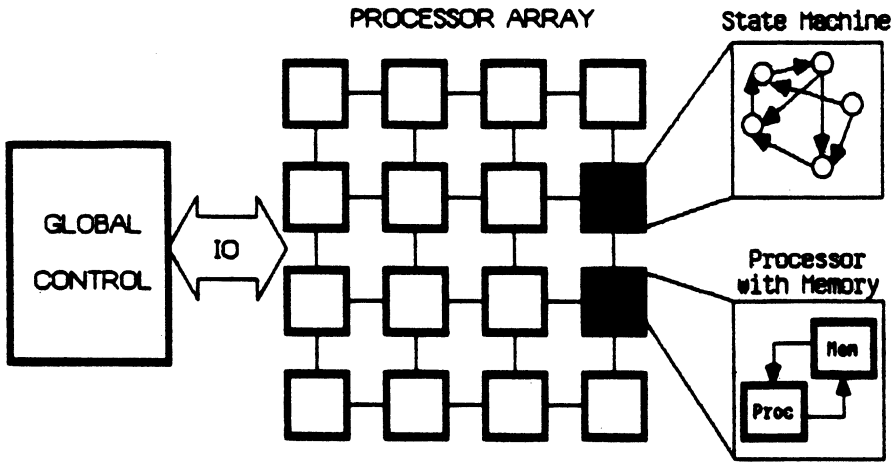


Figure 2.3 Full-Array Structure

processors with storage at each node. Examples include CLIP4, a 96 × 96 array with 32 bits/node [Duff78]; the Distributed Array Processor (DAP), a 64 × 64 array with 4K-bits/node [Ilif82]; the Massively Parallel Processor (MPP), a 128 × 128 array with 1K-bit/node [Batc80]; and the Adaptive Array Processor (AAP), whose basic building block is a single chip 8 × 8 array with 96 bits/node [Aoki82]. Some machines with large memories at each node, e.g., MPP and DAP, incorporate a notion of grid-folding: grids larger than the physical array are folded into several planes and mapped onto the storage available at each node. Some also provide for limited global communication, e.g., DAP includes an additional bus for each row and column, enabling complete row-vectors and column-vectors to be moved around the array.

The subarray class is further subdivided, and is characterized by the range of subarray sizes and the connections between distinct subarrays. A subarray is an array much smaller than the entire grid to be processed; it is simply a processing window. The smallest subarray is a sin-

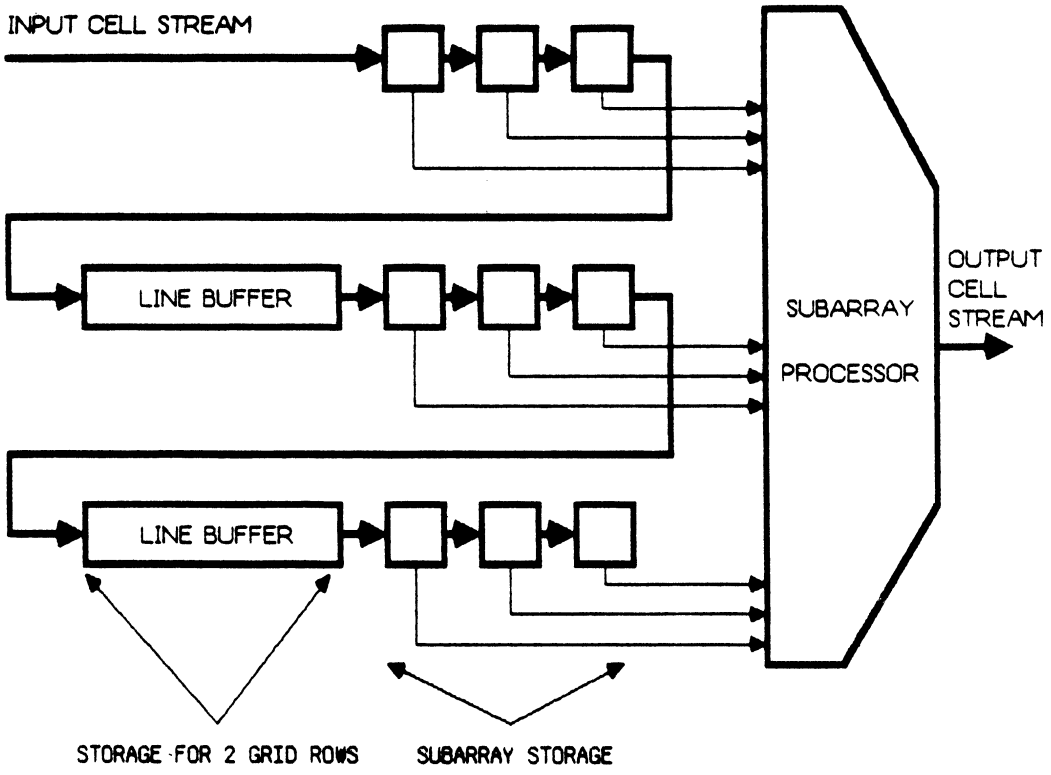


Figure 2.4 Raster Single Subarray Structure

gle neighborhood, 3 × 3 on a square lattice, while the largest is generally between 16 × 16 and 32×32. The simplest subarray organization is the class of *raster, single-subarrays* shown in Fig. 2.4. The idea is to process the entire cell grid in a serial stream (raster order) as it passes by a subarray processor. To do this, shift-registers are introduced as buffers for a few rows of the grid. As the stream passes through the buffers and the processor, enough of the grid is present to insure that each subarray of cells eventually arrives at the subarray processor to be processed. The GLOPR machine [PrNo72] is an early example of this using a hexagonal subarray and operating on a single bit-plane; the Texture Analyzer machine [KlSe72] is a similar contemporary. Although single-subarray systems need not employ a raster structure as in Fig. 2.4, it turns out that subsequent non-raster machines generally use more than one subarray. Hence, this category actually implies a raster organization.

The subarray class is not restricted to a single subarray processing element. The second subclass contains the multiple-subarray machines and is itself subdivided.. Because the single subarrays just discussed output a data stream with format identical to the input stream, it is possible to connect several in a pipeline. Here the individual processors are called stages, and the entire machine is a *raster pipeline subarray* (see Fig. 2.5). This is the organization of the

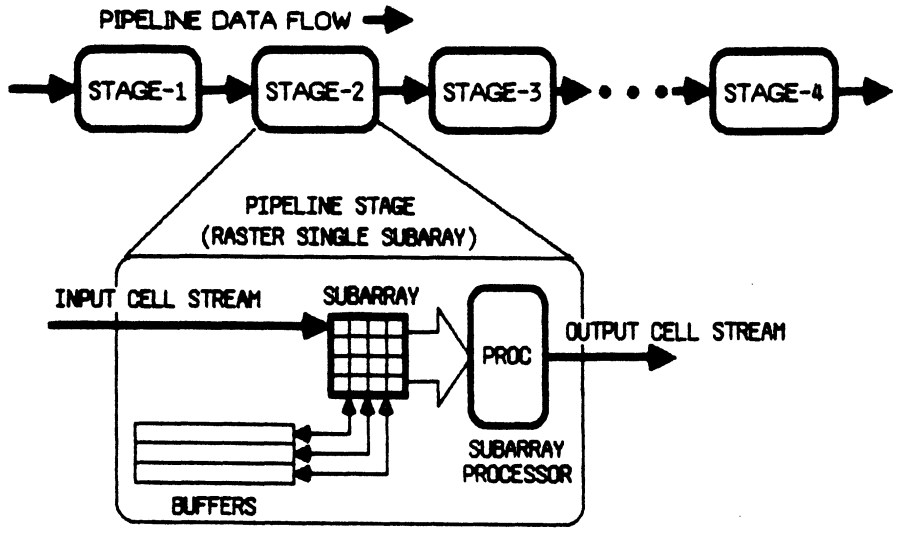


Figure 2.5 Raster Pipeline Subarray Structure

cytocomputer family [LoMc80, LoMS80]. The RPS organization and its antecedents are treated more fully in Chapter III.

Multiple subarrays are not restricted to a raster input format. Non-raster organizations use several interconnected subarray memories and subarray processors to concurrently process several pieces of a complete grid (see Fig 2.6). The major difference between these machines and the apparently similar ICN-based machines is a matter of emphasis. Of primary interest in multiple subarrays is the physical design of the subarray buffers and processors, with their interconnections of secondary interest. In ICN structures much of the architecture is subordinated to the interconnection scheme. This is perhaps the most general subarray structure, since we define it basically in terms of the structure it *lacks*. An early example is the diff3 machine [PDLN79] with four fixed-size binary image memories connected to eight table-driven subarray processors. A modern example is the Preston-Herron Processor (PHP) [HFPS82], in which three cell memories communicate with sixteen table-driven processors.

From this brief survey, some important points emerge. As claimed earlier, not all machines intended for cellular problems have a full-array structure. These alternative structures can be considered to be engineering responses to the problem of constructing an *ideal* full-array--one processor per cell. The engineering constraint is the inability to construct such a

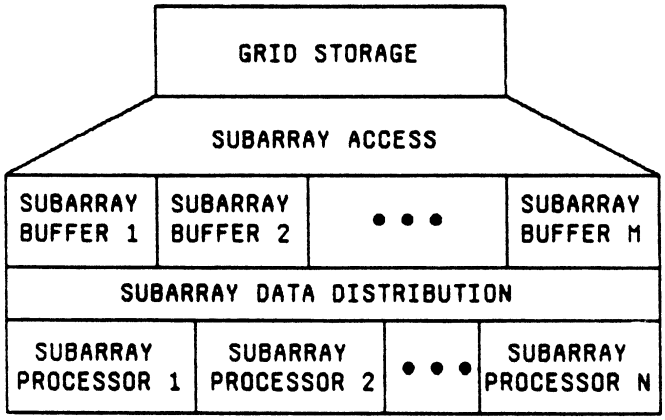


Figure 2.6 Non-Raster Multiple Subarray Structure

large array economically. Thus, the raster single-subarray structure of GLOPR is a very early answer to the problem, conceived at a time when only one subarray processor was technologically feasible. The raster pipeline subarray structure of the cytocomputer family is a subsequent answer, derivable from this earlier structure, conceived at a time when multiple processors became feasible. Non-raster multiple subarrays such as the PHP illustrate another non-array direction made possible when multiple processors became feasible. Hence, we argue that the development of the RPS structure appears as a natural evolution from earlier subarray forms. This is in contrast to other classification schemes [DaLe81, Pres83] in which the RPS cytocomputers are inappropriately categorized as completely disjoint from all other cellular organizations.

2.3. DA Architectures

We turn now from general cellular structures to DA machine structures. This section examines design issues for DA engines and surveys proposed and constructed DA engines. This will provide background necessary for a later evaluation of RPS-structured DA engines.

2.3.1. Design Issues for DA Hardware

This section reviews key issues related to the design of special purpose DA engines. Issues common to the design of standard computing machines are not stressed. Rather, we review the consequences of dedicating hardware to DA tasks.

Adshead [Adsh82a], and Abramovici *et al.* [AbLM82] briefly suggest several positive attributes of successful DA hardware, and some potential drawbacks. Our discussion expands on this theme. We identify seven relevant design issues: task model and architecture, speed and cost-effectiveness, range of application, task representation and limitations, hardware/software boundaries, modularity, and host environment. Although these will be treated separately, the partition is somewhat artificial; design choices in one area force tradeoffs in another. These seven areas can be considered metrics by which to evaluate the performance of a specific DA engine.

- **Task Model and Architecture**

  A task model comprises the data representation and processing steps necessary to solve a problem. For example, switch-level networks and gate-level networks are models for logic simulation; local-area cellular grids, discrete polygons and hierarchical cells are models for DRC. Choice of task model is intimately bound together with choice of DA machine architecture: some models admit more parallelism or concurrency, or permit cleaner data movement, or incur less overhead than others. Trading model complexity (substituting *simpler* models) for speed is common; specific model/architecture tradeoffs are examined in the subsequent survey section.

- **Speed, Cost-Effectiveness**

  The central motivation for migration of DA tasks into hardware is to decrease execution time: moderately large problems become cheap, perhaps interactive; previously intractable problems become manageable; more passes through the design cycle are feasible, permitting a more thorough search of the design space, and yielding a higher chance of success before an irrevocable commitment to physical implementation. Nevertheless, improvement upon the best available (affordable) software time is not the sole motivation. From an engineering standpoint, these systems must also be *cost-effective* to be practical. For example, a hardware system offering significantly more performance than a mainframe computer for the same cost as a mainframe should be considered cost-effective, as should a system offering mainframe performance for the cost of a workstation. Almost any DA algorithm can be accelerated by hardware; we argue that the imposed constraint of cost-effectiveness drastically reduces the number of alternative hardware approaches that will compete successfully.

- **Range of Application**

  The issue of whether to build a machine that handles one problem or a range of problems involves tradeoffs between speed and flexibility. A single-purpose machine can be optimized more fully than a multi-purpose machine. A single purpose machine may need to achieve significantly more performance than the multi-purpose machine if it is to be as cost-effective.

- **Task Representation, Task Limitations**

  A hardware solution may require that the common representation of a DA task (i.e., the software representation) be transformed into something suitable for the hardware. Likewise, the result may require a transformation back into the common representation. These processes must be accounted for if they are time-intensive and can degrade overall performance. There may also exist hard limits on the allowable size of the task that can be processed in hardware, limits imposed by the physical structure of the DA engine. Practical hardware must accommodate both the complexity of current systems and the foreseeable growth of VLSI systems.

- **Hardware/Software Boundaries**

  A common view of design is one of descent through decreasingly abstract, increasingly physical levels in a hierarchy, a progression from behavior toward implementation. In such a model, a DA task is typically a transition between levels of the hierarchy. We might suppose that special DA hardware *replaces* entire steps in the descent, but such a view is an overly restrictive simplification. In a real system it will be necessary to address *during design* the location of the boundaries between the DA engine and its environment, which we assume is mostly software. This is of concern for three reasons. First, not all phases of any DA task necessarily have a structure that justifies hardware implementation. Some phases may have little parallelism or concurrency (e.g., book-keeping operations) and no reasonable fit onto the rest of the DA engine; these may be adequately performed in the host environment. Second, the engine itself may be programmable and require software support from the host environment. Third, the DA engine will in any event need to communicate with other levels in the hierarchy, which are likely to be implemented in software.

- **Modularity**

  Modularity is closely related to questions of range-of-application and task-limitations. A *modular* system can be expanded or augmented, without complete redesign, to accommodate larger or different tasks. This implies some tradeoff against speed: modular systems may incur more overhead than tightly integrated systems that satisfy critical time

constraints for a select set of problems. However, modular systems may be more cost-effective: they need not be redesigned to accommodate increasingly complex problems. Note that modularity applies both to the hardware and software components of a DA engine. If we assume that the primary competitor to a DA engine is a large software package running on mainframe computer, then any software component of the engine must be at least as maintainable as the competing software package. This becomes clearer while reflecting upon the fact that the hardware engine may require arcane and subtle programs (managing parallelism, scheduling, transformation, IO, etc) to succeed.

- **Host Environment**

  Given that DA hardware replaces only portions of a descent-through-levels design, it is clear that a practical DA engine must interact with the rest of the design process. This interaction is loosely what we call the *host environment* and includes: IO overhead required to move problems or results to and from the hardware, programming overhead, and task representation transformation. For example, IO bandwidth between the hardware and its surrounding environment (assume configuration as a peripheral processor) is an important design issue. The advantages of a fast engine may be negated by an inappropriate interface.

Design choices in these seven areas define the engineering of a DA engine--they determine its practical application. Fig. 2.7 illustrates common relationships among these issues, showing a simplified comparison between software and hardware approaches to one DA task. Notice the possible need for representation changes, IO overhead, and an explicit tradeoff of model complexity for speed, yielding a faster solution to a perhaps simplified form of the problem.

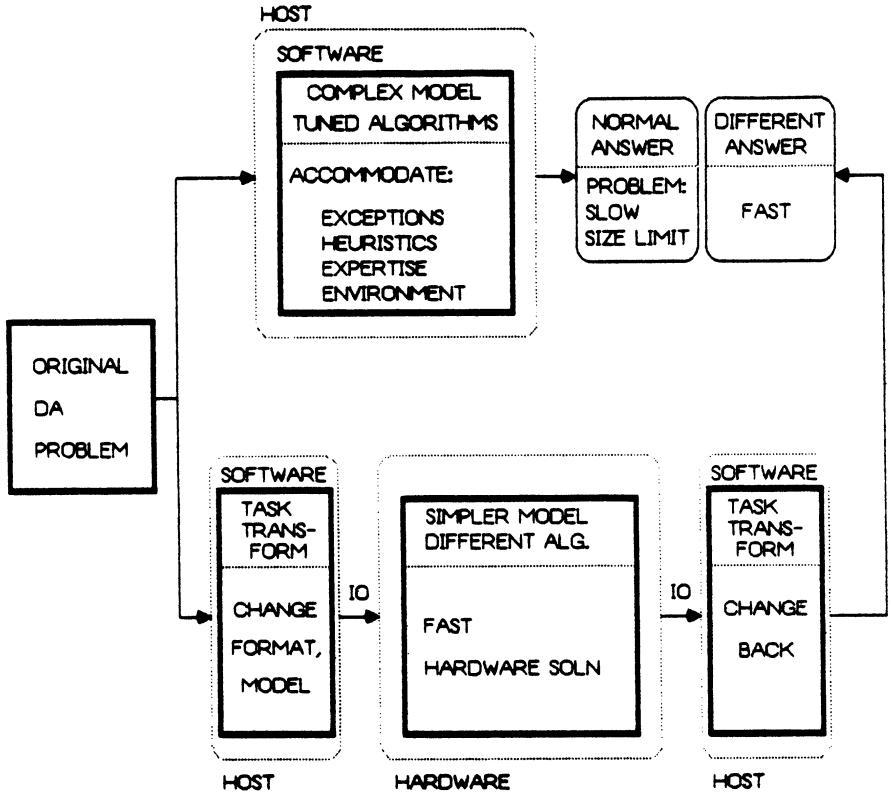


Figure 2.7 Common Relationships between DA Hardware and DA Software

2.3.2. Survey and Critique of DA Engines

This section surveys research on DA hardware and serves two purposes. First, it provides background for a later evaluation of RPS-structured DA engines. Second, it makes concrete the evaluation metrics just discussed by providing examples of the consequences of different design tradeoffs chosen for DA engines.

Considerable DA architecture research has appeared recently, e.g., [AbLM82, Adsh82a, Adsh82b, BaST80, Blan82, BlSv81, BrSh81, Carr81, DaGe82, Dunn84, HoMW83, HoNa83, HoNS81, IoKB83, Iosu80, KaSa83, MRLA82, MuLT81a, MuLT81b, Pfis82, RuMA83a, RuMA83b, RuMA84, Seil82, SKOT83, UeKH83, vanA71, vanB83]. For clarity, we partition this body of work in two ways: level of abstraction, and intended application.

To define levels of abstraction, we observe that not all DA hardware research pertains to concrete system building activities. Rather, a survey of the literature yields a clear pattern to the evolution of DA architectures, starting from abstract notions of parallelism and algorithm restructuring, and proceeding toward concrete mappings onto machine structures and system integration. Hence, the first partition categorizes research by where it falls in this evolution from idea to system. Three phases are identified:

- **Problem Restructuring Studies**

  By *restructuring* we mean the mapping of a standard DA task onto new hardware. This typically begins with identification of intrinsic parallelism in a DA problem or a critical performance bottleneck in an algorithm. The next step is a suggested architectural solution, for example: a machine organization, instruction set, or technology enhancement to an existing structure. The final step is a demonstration that this solution is feasible, usually by simulation of small, well-structured DA tasks running on an idealized machine.

- **Machine Feasibility Studies**

  Engineering issues are considered in this phase. A machine feasibility study implies either a new prototype hardware system, or a retrofit of an existing special-purpose machine that demonstrates the viability of the machine organization for DA tasks.

- **System Integration Studies**

  In this final phase appear full-scale hardware systems interfaced with a complete DA host environment and intended to solve real-world DA problems.

The second partition categorizes DA machines by intended application. Those machines intended for a range of tasks thus appear more than once. This organization permits close comparison of differing architectures solving similar problems.

Each of the following sections reviews research directed toward one DA task. A tabular format is employed which emphasizes the central features of each contribution. Each table gives relevant references for a particular study, summarizes the DA task modeled and any assumptions about its structure, describes the architecture of the DA engine suggested for this task, gives the status of any real implementation efforts, and classifies the entire study by its

level or levels of abstraction. Important examples of design tradeoffs categorized earlier in this chapter are clarified by reference to concrete systems. We defer detailed discussion of the *cellular* aspects of these DA engines to a later section.

2.3.2.1. Routing Engines

Table 2.1 describes research on routing hardware. The important shared characteristic of all these systems is that each implements a maze-router. Maze-routing is attractive for hardware because of the potential for parallelism in the search for paths in a cellular routing grid. All these systems are cellular by our earlier definition.

Moreover, all but two -- [DaGe82] and the system examined in this thesis [RuMA84] -- are full-array organizations.

The L-machine [BrSh81] and the Wire Routing Machine (WRM) [HoNS81] are illustrative examples of different directions for model/architecture tradeoffs. Each is a full-array organization. The L-machine trades model complexity for speed: one-layer unit-cost routing is implemented as a very large array (1024×1024) of simple state-machines for marking the grid. This is an elegant approach, fully exploiting the parallelism of maze-routing. Practical utility, however, does not necessarily follow. Production-quality maze-routing software often employs complex path-functions and heuristics [Souk81]--techniques not easily captured in this simple model--to achieve acceptable performance. On the other hand, the WRM implements a global maze-routing algorithm of *greater* complexity than common software approaches, and uses special hardware to accelerate the execution speed. The algorithm [NHLV82] requires general processing elements in the array, in this case microprocessors, and uses a small, virtual array onto which is folded a larger routing grid. However, the performance improvements on the small 8×8 WRM prototype were at best modest: basic problems required roughly 1 minute to execute on either the WRM or a mainframe computer. Assuming that a few minutes of mainframe computer time is easily obtained and is not a burdensome delay, the added complexity of managing special hardware can detract from its acceptance, *despite* the hardware's obvious cost-effectiveness.

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[BlSv81], [Blan82]	Maze-routing for detailed wiring. Unit-cost grid; 1-2 layers. BMP has fixed size; large Virtual grids folded onto physical array of VBMP.	Bit Map Proc (BMP): Fixed 1024×1024 array of bit-serial proc's + mem. Each array node addressable. Virtual BMP (VBMP): 32×32 array of bit-serial proc + mem. Support for folding large grids onto physical array. Each array node addressable.	BMP: Proposed VBMP: 4×4 nodes under construction at Stanford Univ.	BMP: Prob. Restr. VBMP: Mach. Feas.
[HoNS81], [NHLV82], [HoNa83] (also [Bald83])	Maze routing for global wiring. Weighted grid with expected wiring cost/demand; 2 layers. Detailed wiring proposed. Large grids folded onto physical array.	Wire Routing Machine (WRM): 32×32 array (max) of μproc with mem. Support for folding large grid onto array. Each array node addressable.	Functional: 8×8 array with Z80 μproc, 15Kb mem/node	Mach. Feas., Sys. Integ.
[Adsh82a], [Adsh82b]	Maze routing for detailed wiring. Unit cost grid, 2 layers. Large grids folded onto physical array.	Distributed Array Processor (DAP): 64×64 array of bit-serial proc + mem. Global row/col busses for global movement. Minicomp. host.	DAP is commercial machine	Mach. Feas. Sys. Integ.
[Carr81]	Maze routing for detailed wiring. Unit cost grid, 2 layers. Fixed grid.	Smart Memory Array Processor: memory array with logic on one chip.	Chips designed	Prob. Restr. Mach. Feas.

Table 2.1 Survey of Routing Engines

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[BrSh81]	Maze routing for detailed wiring. Unit cost grid, 1 layer. Fixed size grid.	L-Machine: 1024×1024 fixed full-array of L-cells; L-cell is state automaton for marking grid.	Proposed	Prob. Restr.
[Iosu80]	Maze-routing for detailed wiring. Unit cost grid, 1 layer.	Modular L-Machine: Partition full-array of [BrSh81] into smaller subarray chips which communicate serially to reduce pin count.	Proposed	Prob. Restr.
[DaGe82]	Maze routing for detailed wiring. Weighted grid, 2 layer PWB's.	CAD-CALAY: Minicomp. plus: Routing-Kernel in hardware Extra μcode Cell memory Extra busses	Functional: LSI 11/23 + Hardware kernel	Mach. Feas.
[RuMA84], [RuMA83a], [RuMA83b], [MRLA82]	Maze routing for detailed/global wiring. Unit cost grid, 1-2 layers. Fixed grid.	Raster Pipeline Subarray Cytocomputer: pipeline of 3×3 subarray stages, minicomputer host.	Functional: employ prototype for commercial machine	Mach. Feas. Sys. Integ.

Table 2.1 continued

2.3.2.2. DRC Engines

Table 2.2 describes research on design rule checking hardware.

All except one system [KaSa83] employ a local-area cellular model: masks are represented as bit-planes on a large uniform grid. Design rules are local in extent, usually checked inside a small processing window. This trades complexity for speed: no electrical parameters are extracted; some incorrect errors are flagged due to connectivity problems; arbitrary obliques are not permitted. But the flattened, uniform mask representation is conducive to rapid local processing of cells.

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[BlSv81], [Blan82]	Local area DRC: mask represented as cellular grid, Manhattan geometry only. Large grid folded onto processor array; rectangles can be loaded into array by individually addressing rows/cols in array. Errors marked in grid.	BMP/VBMP: see routing entries	BMP: Proposed VBMP: 4×4 nodes under construction at Stanford Univ.	BMP: Prob. Restr. VBMP: Mach. Feas.,
[Seil82]	Local area DRC: mask represented as cellular grid, Manhattan and some 45-degree geometry. Large grid streamed through processor in raster order; rasterization done elsewhere; error locations sent back to host. Width, edge, boolean mask ops are hardware primitives.	Hardware Assisted DRC: Raster single subarray organization. Line buffers and custom chips for: Width-check Edge-check Derived mask check (Boolean ops) Rasterizer μcomp. controller Minicomp. host	Functional: Custom NMOS chips	Mach. Feas., Sys. Integ.
[RuMA84], [RuMA83a], [MRLA82], [MuLT81a], [MuLT81b]	Local area DRC: mask represented as cellular grid, Manhattan geometry only. Large grid streamed through pipeline in raster order; rasterization done elsewhere; errors marked in grid. Implements tolerance checks on shapes in parallel in pipeline stages.	Raster Pipeline Subarray cytocomputer: pipeline of 3×3 subarray stages, minicomputer host	Functional: employ prototype for commercial machine	Mach. Feas.,
[KaSa83]	Mask represented by edges of features. Width/space checks on well-formed polygons with Manhattan geometry.	Systolic DRC (SDRC): 2 systolic sort arrays (SAX, SAY) for horiz, vert. edges. Linear systolic DRC array for width/space checks in 1 dimension (horiz or vert).	Proposed	Prob. Restr.

Table 2.2 Survey of DRC Engines

2.3.2.3. Placement Engines

Table 2.3 describes research on placement hardware.

All these systems attempt to minimize the (possibly weighted) sum of wiring lengths between connected modules. Although this is a common model for placement, experience with large problems [Souk81] has recognized that minimum wiring length alone is insufficient to guarantee wirability, and that more complex complex metrics, e.g., weighted sum of wire lengths *and* maximum channel congestion, produce better results. Hence, these machines also trade some model complexity for speed.

Moreover, the similar problem restructuring for a full-array suggested by [UeKH83] and [ChBr83] presents several difficult engineering problems. The model is based on parallel inter-

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[UeKH83]	Parallel pairwise interchange placement of connected modules to minimize total wiring length. Modules are vertices of graph to be embedded in a grid. Fixed size grid.	Full-array of processors with memory. Global bus connects all nodes. Global controller.	Proposed	Prob. Restr.
[ChBr83]	Parallel pairwise interchange placement of connected modules to minimize total wiring length. Modules are vertices of graph to be embedded in a grid. Fixed size grid.	Full-array of processors. Processors have a Content Addressable Mem to store connected modules; tally-circuit to count modules in CAM connected to module in cell. Global bus connects to all nodes. Global controller.	Proposed	Prob. Restr.
[IoKB83]	Pairwise interchange of connected modules to minimize total wiring length. Solve quadratic assignment problem (min sum of module distances over weighted nets) incrementally after each interchange.	Module Interchange Placement Machine: 4-stage arith. pipe (solve QAP) Connections table (Mem nets/mods) μcomp controller Minicomp. host	Functional: 256 modules max	Mach. Feas.,

Table 2.3 Survey of Placement Engines

change of adjacent modules in a grid. [ChBr83] suggests a Content Addressable Memory (CAM) in each array-node to store module connectivity information.[1] This presents problems for highly interconnected modules such as those on busses or clock-lines: a small CAM will not accommodate dense connectivity; a large CAM will be expensive and sparsely occupied over most array nodes. Questions of how to address realistically large problems--the problems that may justify special hardware--on a physically small array have yet to be addressed. Indeed, simulations that verify the technique for such large problems (10^3 to 10^4 modules) have yet to appear in the literature. This work is an example of the difficulty in transforming elegant ideas about restructuring into practical engineering systems.

2.3.2.4. Simulation Engines

Table 2.4 describes research on simulation hardware.

This has been the most active area of research in special DA hardware due to demonstrated large performance-gains achieved with special hardware. Two approaches have been investigated: event-driven simulation, in which time is quantized and components are only updated when they are scheduled to change, and compiled-code simulation, in which the network to be simulated is compiled into an executable description that is independent of network inputs. Table 2.4 shows that event-driven systems have been more numerous.

The evolving family of compiled-code simulators developed at IBM--LSM, YSE, EVE--illustrates serious, large-scale issues in system-integration. For example, the YSE is currently constrained by the IO bandwidth of its host connection [BHST84]. Capturing simulation trace data thus takes place at the slower host speeds, rather than native YSE speed, and the overhead makes the YSE *slower* than a mainframe when applied to modest problems, despite its superior performance on large problems [Dunn84]. [PfKr82] and [KMMN83, HKMR83] discuss the task-transformation software and user-interface software necessary to support the compiled-code model on the YSE and LSM respectively. This work also exemplifies some range-of-application tradeoffs. Although easily regarded as a gate-level simulator, [BHST84] shows how MOS pass-

[1] See [WeLF82] for a discussion of CAMs in array structures.

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[vanA71]	Event driven gate simulation	Boeing Computer Simulator	Functional: 48K gates max	Mach. Feas., Sys. Integ.
[HoMW83], [BLMR83], [KMMN83], [HKMR83]	Compiled gate simulation (non-event driven); every gate evaluated every cycle: Variable gate delay; a gate is a 6-input 1-output function over 2-bit values. Software schedules processing over a crossbar.	Logic Simulation Machine (LSM): 1 control proc. logic proc's (eval. logic expr) array proc's (sim. memory) 64×64 crossbar (communication) mini. interface mainframe host	Functional: 64512 gates max	Mach. Feas., Sys. Integ.
[Pfis82], [Denn82], [PfKr82]	Same as LSM, except: unit gate delay and rank-order simulation (~ zero gate-delay); a gate is a 4-input 1-output function over 2-bit values.	Yorktown Simulat. Engine (YSE): 1 control proc. logic proc's (eval. logic expr) 16×16 crossbar (communication) bus bus-controller (comm. all proc) μcomp. interface mainframe host	Functional: 16 logic proc. 0 array proc. 64K gates max	Sys. Integ.
[Dunn84]	Same as YSE	Engin. Verification Engine (EVE): same as YSE but 256×256 crossbar, 2.0M gates max, 12.8Mb array memory.	Under construction at IBM	Sys. Integ.
[BaST80], (also [AbLM82])	Event driven gate simulation. A gate has any number of fan-ins or fan-outs; all gates examined each time-unit for event activity.	Tegas Accelerator: 5 memories: Status&Data (net descrip) Fan-In Fan-Out Activity Flags (active gates) Event Queue (schedule events) 3 proc. units: Evaluation (8 identical units for gate eval) Event Queue Update (7-stage pipe)	Prototype under construction, 32K gates max.	Mach. Feas.

Table 2.4 Survey of Simulation Engines

References	Model and Assumptions	Architecture	Engineering Status	Level of Abstraction
[GiCa83]	Event driven simulation. Gate-level simulation with 12 logic values. Functional and behavioral simulation using macros.	Megalogician: 3 processors in pipeline ring: Queue unit State unit Evaluation unit (Each unit has dedicated memory)	Commercial machine: 1M gates max	Sys. Integ.
[vanB83] (also [Meye84])	Event driven gate simulation; 3-input gates.	ZYCAD Logic Evaluator: multiple memories, processors, and pipelines.	Commercial machines, max gate capacity: LE-1001: 50K gate LE-1002: 100K gate	Sys. Integ.
[Adsh82a]	Event driven gate simulation: map logic primitives into 2D sheets; fold sheets onto physical array processor; net connections modeled as stored node addresses in each sheet; sweep through sheets to process events.	Distributed Array Processor (DAP): see Routing entry.	DAP is commercial machine	Prob. Restr., Mach. Feas.
[SKOT83]	Event driven gate simulation: simulate "blocks" of logic; block is $\sim$ 10-100 gates or 1-10 RAMs; two-valued logic.	HAL: 1 control proc. logic proc's (sim. blocks) memory proc's (sim memory) 32×32 multistage-network minicomp. host	Functional: 29 logic proc. 2 mem proc. 1.5M gates max 2.5Mb mem max	Sys. Integ.
[AbLM82]	Event driven simulation of simple elements (gates, flip-flops) or functional elements (high-level blocks). Events are messages that travel through processing pipeline.	Logic Sim. Machine: Pipeline ring with FIFO's between stages (dataflow). μprog. processors: Current Event Model Access (get fanout) Schedule (delays) Event List Manag. Simple Config. (eval gates) Functional Eval. (eval funct. elems in parallel proc's) Memories: Event List Model Data (network descr.)	Proposed	Prob. Restr.

Table 2.4 continued

transistor networks can be handled by the YSE with appropriate transformation software. This is possible because the YSE is a programmable machine, executing 4-input 1-output table-lookup instructions (i.e., abstract gate evaluations). Several YSE instructions are required to evaluate a transistor (as opposed to one for a simple gate) thus trading some speed and problem-size for added flexibility.

2.4. DA Architectures as Cellular Architectures

This section connects the general cellular architectures of Sec. 2.2 with the DA architectures of Sec. 2.3. This thesis suggests that the RPS structure, which evolved outside the DA area in picture-processing applications, is a novel, useful model for DA engines. Before we can examine this claim in detail, three fundamental questions must be answered:

- Are there DA problems which are cellular problems?
- Are there cellular DA machines that address these problems?
- Do all such DA machines necessarily employ identical machine architectures? That is, are they all derived from a standard model such as the full-array?

The answer to each of the first two questions is clearly positive: note the variety of grid-based DA models and cellular DA machines surveyed in Sec. 2.3. Cellular machines can certainly tackle DA tasks. The answer to the last question is negative: we find a variety of *different* machine organizations among the cellular DA engines surveyed. It is important to observe, however, that these various cellular DA engines are *not* unrelated: many of these machines are neatly categorized by the simple cellular taxonomy developed in Sec. 2.2. This idea is made clear by stepping through the taxonomy of Fig. 2.1 and identifying the relevant DA engines.

Full-array structures have been particularly popular for DA. Breuer and Shamsa [BrSh81] have proposed a single chip 256 × 256 array of finite-state machines (an L-machine) to perform unit-cost Lee routing and a multi-chip 1024 × 1024 machine. Iosupovicz [Iosu80] discusses the details of such a routing machine based on an interconnection of smaller, more modular building block chips.

Adshead [Adsh82a, Adsh82b] has reported successful application of the DAP machine to problems in maze-routing and logic simulation. Routing involves folding large grids onto the physical array and attending to edge effects. Simulation involves coordinating updates to logic gates distributed across the nodes in the array.

Blank [BlSv81, Blan82] has proposed two array architectures for solving general bit-map DA problems: a Bit Map Processor (BMP) and a Virtual Bit Map Processor (VBMP). A BMP is a standard array of 1024 $\times$ 1024 simple processors each with memory. A VBMP is a much smaller array, e.g., 32 $\times$ 32, with special hardware to fold a larger virtual grid onto the physical array. Large memories reside at each array node, and the edge and corner nodes include special mechanisms for dealing with border effects. Each cell in the array can also be individually addressed via row and column lines to provide some global communication. Simulations for DRC and simple maze-routing have been constructed, and a 4 $\times$ 4 TTL prototype is being fabricated.

Hong *et al.* [HoNS81, HoNa83] describe a Wire Routing Machine based on an array of microprocessors, which also incorporates provisions for folding large problems onto the array. An 8 $\times$ 8 prototype with 15K bytes/node is operational, and claims are made that a 32 $\times$ 32 structure would likely suffice for all real problems. Complex global-routing algorithms have been implemented and run on modest test grids [NHLV82].

Placement algorithms have also been considered in a full-array environment. The basic idea is to perform many concurrent pairwise interchanges among adjacent modules until total wire length is minimized. The restriction to adjacent interchanges enables each node to compute the change in wire length from one interchange; the array structure enables concurrent interchanges. Ueda *et al.* [UeKH83], and Chyan and Breuer [ChBr83] describe similar array machines for placement.

Bus-structured machines have also been constructed. Damm and Gethoeffer [DaGe82] have built a Lee-routing engine by modifying a commercial minicomputer. A special cell-memory, a hardware "kernel" of routing operations, and an interconnecting bus were added to optimize performance. Successful operation with PC boards has been reported.

Subarray architectures also appear. Seiler [Seil82] has developed a hardware implementation of Baker's raster DRC [Bake80] using a raster single-subarray. The processing section uses a few custom PLA-based chips to perform width checks, edge checks, and logical combination of mask layers in a small window. A feedback mechanism with shrink/expand templates is provided in the subarray to enable larger width checking using multiple passes through the processor. An RPS-structured cytocomputer [MuLT81a, MuLT81b, MRLA82, RuMA83a, RuMA83b, RuMA84] has been used to perform DRC and routing; these algorithms are the subject of subsequent Chapters.

(Outside the area of grid-based DA, several machines have been studied which also fit our taxonomy. Kane and Sahni [KaSa83] describe a systolic[2] array organization for DRC using polygon edges as the basic data element. The Logic Simulation Machine [HoMW83], the Yorktown Simulation Engine [Pfis82], and the Engineering Verification Engine [Dunn84] are compiled-code logic-simulators developed at IBM, all with an ICN structure. From 16 to 256 logic processors, each storing and updating logic elements, communicate over a cross-bar switch.)

The ability to categorize these systems neatly as specific instances of more general cellular structures supports our premise that cellular organizations are appropriate models for DA engines. Indeed, the approach to DA hardware taken in this thesis is suggested by this analysis. Instead of attacking a single DA task with one specific, tightly optimized machine architecture, we shall examine the relevance of a general *class* of cellular architectures to particular DA tasks. There is one striking advantage to this general-to-specific approach: because we start with a broader view of basic cellular architectures before we pursue specific applications, we can avoid reinventing solutions that arose in (apparently) dissimilar applications.

Given the connections made in this section, we refine the objectives for this thesis given in Chapter I. This chapter has shown that many different cellular organizations have been used as basic models for DA machines. The traditional parallel array structure has been particularly popular as a *paradigm* for DA machines. Research in this area provides an apt analogy to our

[2]Systolic structures [Kung79] are generally labeled as cellular; it is only our nonstandard definition of the term *cellular* to mean "applicable to grids" that labels them otherwise.

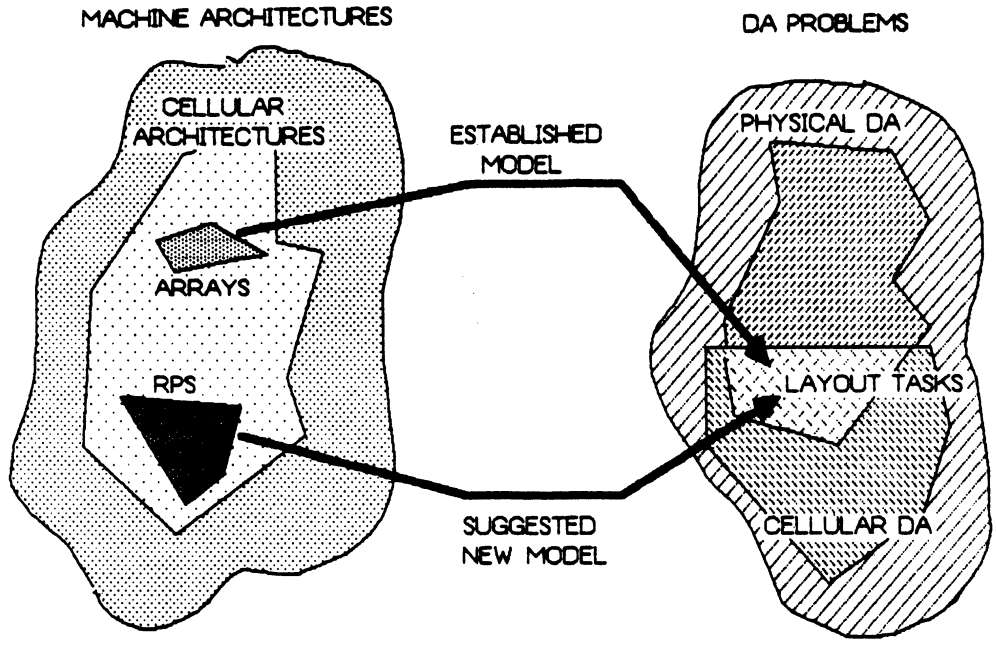


Figure 2.8 Cellular Hardware Paradigms for DA

own goals (Fig. 2.8): this research examines engineering issues related to the design, analysis, and optimization of DA engines that use an RPS organization as the basic model.

2.5. Summary

This chapter connects research on general cellular structures, such as those developed in classical picture-processing applications, with research on DA engines. The RPS organization is introduced and its relationship to other general cellular structures is clarified. Engineering issues in the design of practical DA engines are examined; a DA hardware survey shows that several DA engines have been applied to grid-based DA problems, and that these diverse engines can be classified as instances of general cellular organizations. This is taken to imply that cellular structures are good candidates for some DA problems. It is argued that the study of DA hardware for grid-based tasks might reasonably be approached by employing a general cellular architecture--the RPS organization--as a basic model for DA engines.

CHAPTER III

RPS ARCHITECTURES AS DA ENGINES

3.1. Introduction

Chapter II introduced the RPS class as an evolution from earlier subarray structures and placed it in the spectrum of related cellular structures. This chapter undertakes a detailed analysis of the RPS organization. The following section examines more carefully the origins of the concept of an RPS class. It defines the class and summarizes some important antecedent ideas related to the class. Issues related to basic design and performance analysis are addressed next. This examination includes a qualitative evaluation of RPS structures as DA engines using the metrics developed in the previous chapter. A quantitative performance comparison for illustrative RPS and array structures is also presented. Finally, we describe the prototype RPS system developed to support DA experiments for this thesis.

3.2. The RPS Class: Origins and Antecedents

The label *Raster Pipeline Subarray* describes the central features of a particular class of machines: these machines perform local computations on a cellular grid as cells stream serially through a pipeline of subarray processors. More succinctly, we shall refer to this idea as *pipelined serial subarray computation.* This idea forms the template for all machines in the RPS class. The following sections examine the origins of the class, and summarize the important cellular tasks and machines that motivate it.

3.2.1. Origins

We discriminate here between two definitions of the RPS class in order to clarify its origin relative to the goals of this thesis. These two definitions are labeled simply *bottom-up* and *top-down.*

The bottom-up definition reflects our early experimental work:

> The RPS class is the appropriate abstraction of the novel features introduced by machines such as the cytocomputers, i.e., pipelined serial subarray computation.

The class is based on two fundamental innovations originally disclosed as patents. The first is the notion of a single, serial subarray processor that performs array-style neighborhood operations without a complete array of processors. This idea was disclosed in a patent issued to Golay *et al.* [Gola65]. The second innovation is the notion of a pipeline of these serial subarray processors which we attribute to the patent of Sternberg [Ster79b]. Because these two patents actually propose picture-processing hardware, they disclose several other ideas related to the implementation of such systems. However, we argue that it is only the idea of pipelined serial subarray computation that is compelling as a general architectural feature[1]. We choose to regard most other issues as application-specific details.

The RPS class was proposed following our early development of DA applications on such picture-processing architectures. [MuLT81a, MuLT81b] first suggested the application of cytocomputers to elementary DRC for Mead/Conway rules [MeCo80]. This was extended to more general cellular DA tasks such as maze-routing in [MRLA82]. As these early machine-feasibility studies progressed toward system-integration research it became increasingly clear that cytocomputer architecture *per se* was an inappropriate focus for research intent on producing practical DA machine structures. It was concluded that the idea of a DA engine designed as an elaboration of cytocomputer architecture was necessarily sub-optimal. Such an approach increases hardware complexity to compensate for an inadequate foundation: the specifics of cytocomputer hardware are optimized to support picture processing, not DA. The alternative

[1] Machines based upon the Golay patent include CELLSCAN and GLOPR (see [PDLN79]). These machines are raster single subarrays by our classification. Machines based on the Sternberg patent include the cytocomputer family; these are RPS machines. The lack of descriptive architectural detail conveyed by some proprietary names motivates the label *Raster Pipeline Subarray*, which attempts to describe the structure of pipelined serial subarray hardware.

proposed in [RuMA83a, RuMA84] is to abstract a class of machines embodying only a pipelined serial subarray organization. The name Raster Pipeline Subarray for this class was suggested in [RuMA83a]. Specific machine implementations are regarded as members of the class optimized for specific problems. This thesis uses the class as the point of departure from previous picture-processing research toward new applications such as DA.

On the other hand, the top-down definition ignores the insights gained from these early problem-restructuring and machine-feasibility studies. The premise here is simply to accept pipelined serial subarray structures as the basic template for the RPS class, and to describe this structure in current terminology:

> The RPS organization is a systolic structure. Local (subarray) operations on a cellular grid are spatially distributed across processors arranged in a linear pipeline.

We take *systolic* to mean spatial-pipelining [Kung79]. The necessity of cell-buffering hardware to properly align successive subarrays of data in the cell-stream that moves through each processor does not affect the basic classification as a systolic structure [Kung84]. Hence, the RPS class is a proper sub-class of linear systolic arrays. The bottom-up definition suggests more of the original motivation for the class, while the top-down definition is a more succinct description of the template for the RPS class.

3.2.2. Antecedents

The taxonomy of Chapter II focussed on the architectures of related cellular machines and largely ignored the algorithms they performed. This section briefly surveys the tasks and algorithms that influenced the development of the RPS structure. We view cellular machines as special-purpose engines built to solve particular cellular problems. Following the previous development, we look mostly at picture-processing applications. Nevertheless, it turns out that ideas developed here are influential beyond this restricted application. Three areas are examined: cellular automata, binary template-matching, and mathematical morphology. These three areas have been especially influential to cytocomputer architecture, and thus indirectly to the RPS class.

Studies of cellular automata are a starting point for most subsequent studies of cellular structures [Burk70]. A general form is a cellular space of automata, which commonly appears as a collection of locally-connected state-machines embedded in a two-dimensional lattice. Even at this level of abstraction some precursors to picture-processing appear [Ulam70]. For our purposes though, the important concept is that of an array of state-machines operating in lock-step. An algorithm here is defined by the state-machine at each lattice point, and the problem input is an initial set of machine states at each lattice point. Likewise, the output is the aggregate state arrived at after several synchronized state transitions of each machine in the grid.

The second operation to consider is template-matching on binary images. Given one binary image (the input) and a small binary template, we produce a new binary image with a 1 in each cell where the template matches the configuration in the input image, and count the number of template matches. The match-and-count approach motivates several real machines, notably CELLSCAN and GLOPR, the first raster single-subarrays. Several global topological properties of binary images can be ascertained using only local match-and-count; see [Gola69, Gray71].

The final class of operations we review is called mathematical morphology, originated in [Math75, Serr81]. Although this formalism comprises a broad theory for the statistical and structural analysis of images, we examine only those aspects influential to machine architecture. Mathematical morphology provides a set of formal operators and an algebraic framework in which to manipulate them. Local template matching again forms the basis of these operators. Additional detail is provided here because this approach has a substantial impact on cytocomputer architecture, and because the notation and some of these ideas will appear later in our DA applications.

Again, we begin with binary images. In this formalism, a binary image is represented as a set of points, for example, black foreground points on a white background. A point is an ordered pair of coordinates in a two-dimensional space, i.e., (x, y) coordinates in a grid or continuous plane. If A and B are binary images, and hence sets, the usual image-to-image Boolean operators such as AND, OR, etc. appear in set-theoretic form as intersections, unions, etc.

Three primitive operators and two composite operators suffice to illustrate most of the useful properties of the formalism. These are: translation, dilation, erosion, opening, and closing.

If A is an image, and thus a set, and p is a point in some space, then the *translation* of set A by point p is defined as $A_p = \{a+p \mid a \in A\}$ and is itself another set. If A is regarded as a geometric shape drawn in its own coordinate space with a local origin, then A_p is shape A translated so that its local origin is at point p. With translation define the two primitives of *dilation* $\oplus$, and *erosion* $\ominus$ as follows:

$$A \oplus B = \bigcup_{b \in B} A_b \qquad A \ominus B = \{p \mid B_p \subseteq A\}$$

The dilation $A \oplus B$ is the union of translations of A by points from B. The erosion $A \ominus B$ is the set of points to which we can translate B and still have it contained in A. Loosely, dilation and erosion are formal generalizations of the intuitive ideas of expanding and shrinking. However, erosions and dilations are defined for arbitrarily complex sets A and B, whereas expands and shrinks are usually specified with simple patterns.

Two operators defined as compositions of the dilation and erosion primitives are also useful. These are called *opening* and *closing*. If X and S are sets, X opened by S is $X_S = (X \ominus S) \oplus S$, and X closed by S is $X^S = (X \oplus S) \ominus S$. Again interpret sets X and S as geometric figures. Then X opened by S is the set of points in X touched by shape S as S slides around inside X. Closing has a similar interpretation for the complement of X. Fig. 3.1 illustrates all these operators for figures drawn on a continuous plane.

This formalism motivates special hardware because these operators are *local*, and can be reduced to a sequence of template-matches. In an expression such as $A \ominus B$, A is typically a complete mask, and B is a small figure such as a circle or rectangle[2]. $A \ominus B$ is easily seen from its definition to be a form of template-matching: the result is the set of points at which B matches A. Within the algebraic framework for these operators, duality relations exist that allow the dilation $A \oplus B$ to be regarded similarly. Since the opening A_B and closing A^B are composite operators built upon dilation and erosion, they also reduce to general template-

[2] B is properly called a *structuring element* in this formalism. Operations between image A and such structuring elements are intended to probe the structure of A. See [Serr81, Ster83] for examples.

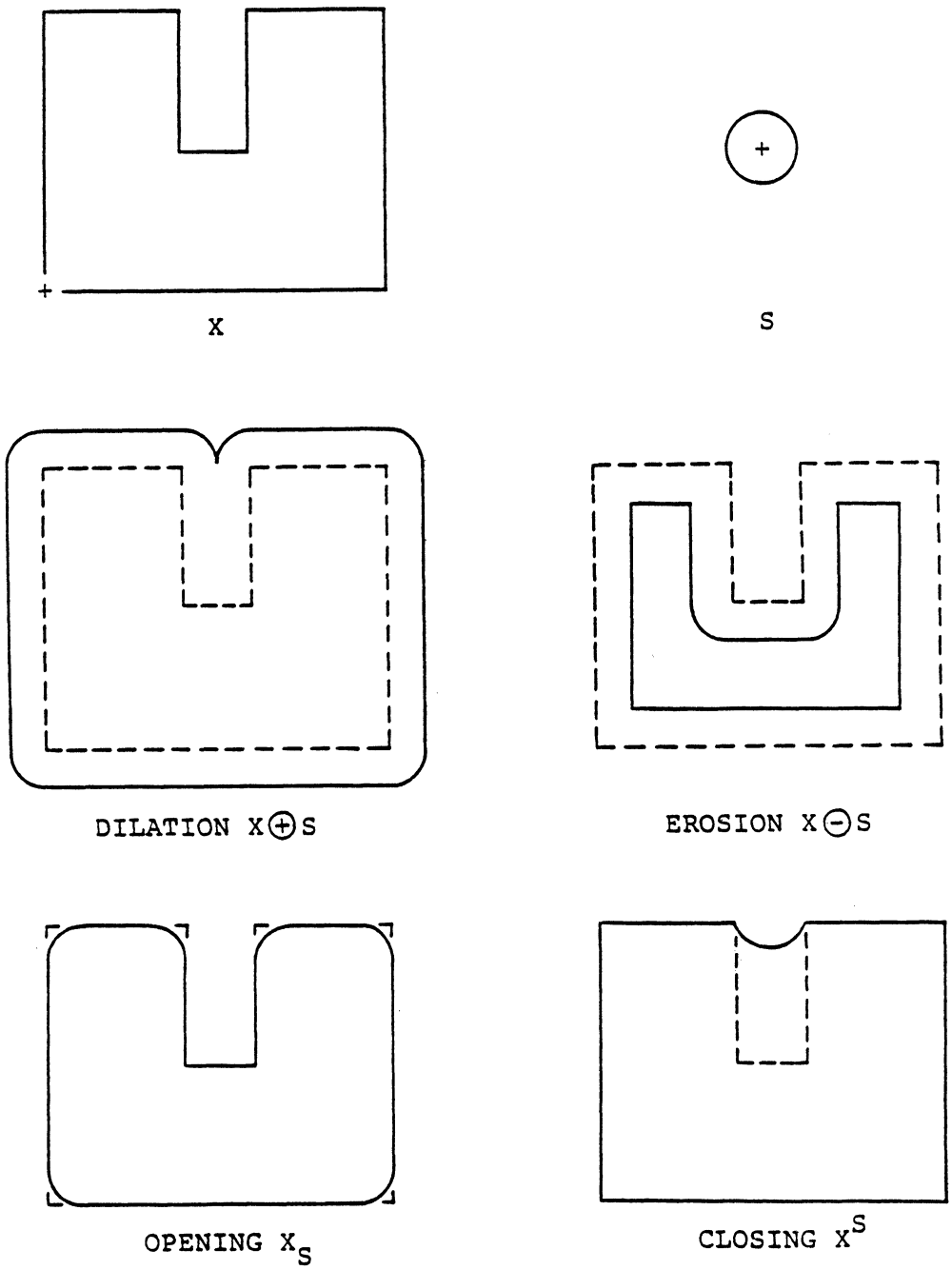


Figure 3.1 Basic Morphological Operators

Shape X operated upon by shape S. Local origins marked '+' for each shape. Result shown with solid-lines, superimposed on original X in dotted-lines.

matching. If B is not a simple figure, but rather a large, complex shape, it may be possible to decompose B into simpler shapes. The algebra includes identities which permit simplifications similar to those done in Boolean algebra. For example, to compute $A \ominus B$ when B is a dilation of simpler shapes $B = B_1 \oplus B_2 \oplus B_3$, it suffices to compute $(((A \ominus B_1) \ominus B_2) \ominus B_3)$ which is a sequence of simpler erosions.

These operators can also be extended from binary images to images with integer-valued cells, generally called grey-level images. Sternberg [Ster80a, Ster80b] has shown that grey-level images can be regarded as sets of coordinate points in a three-dimensional space. Such images are regarded as surfaces, with individual cell values regarded as elevations. Dilations and erosions, and hence any composite operators, can be formulated for this case. In the expressions $A \ominus B$ and $A \oplus B$, B is now itself a grey-level surface, and application of an operator changes image A's grey-level values. [Ster83] shows that these operators can be formulated as local template-matches requiring only simple addition and subtraction on individual cells, and maximum/minimum detection on local cell-groups.

The goal of developing DA tasks on cytocomputers motivated the abstraction of the RPS class. The review in this section answers the related question of what tasks motivated cytocomputer structure:

> Cytocomputer structure is a direct hardware embodiment of binary and grey-level morphological operators using a general template-matching architecture; it is intended to emulate the basic action of state-transitions in a space of automata.

This is verified by studying how the stages and pipeline relate to the morphological operators.

Consider a general operator applied to grid A, A **op** B, where B is a complex figure. To implement in hardware, first use the algebra to decompose B into simpler figures to yield:

$$A \,\mathbf{op}\, B = A \;\mathbf{op}_1\; B_1 \;\mathbf{op}_2\; B_2 \;\cdots\; \mathbf{op}_k\; B_k$$

The goal is to have each B_i fit inside the subarray of each subarray-stage in the pipeline. Because each operator reduces to to a local template-match, one pipeline stage can perform "$\mathbf{op}_i\ B_i$" on the intermediate grid flowing through the stage. Thus, A enters the pipeline and $A \,\mathbf{op}\, B$ exits. Pipelined subarray computation is a way to implement *in parallel* the composition of primitives needed for these image operators. Indeed, cytocomputers have been referred to

as "operator parallel machines" [DaLe81].

Cytocomputer stage architecture bears a strong analogy to the action of state-change in a cellular space. Consider a grid streaming through one subarray stage; at each time step, one cell enters and one cell exits. The exiting cell is a computed function of whatever information is stored in the subarray-stage. Because the entire grid flows through the stage, information can accumulate in the stage to affect future computations. After the last cell has exited, the stage might contain a residue of information computed over all grid cells. Clearly, nothing precludes such computations in a general RPS stage. However, cytocomputer architecture imposes two severe restrictions on such computation: exiting cells are computed only from the state of the current subarray-storage, and no computed information is ever saved in a stage. Hence, computed cells are never retained in a stage and cannot affect future stage computations. Global residues are impossible to compute (for example, in the match-and-count approach, *match* can be done but *count* cannot).

Such a stage design is precisely the model of computation embodied in an array of state-machines. One pass of a grid through one subarray stage is intended to emulate one state-transition in the array. In one time-step, all nodes in the array examine their neighbors and all simultaneously change state; no information ever moves further in one step than the immediate neighbors of one node. Similarly, these stage restrictions mean that no information ever moves further in one stage computation than the diameter of a single subarray; each cell is recomputed as a function of its current subarray neighbors and nothing else. This suffices for the local template-matching required for the image operators.

This discussion clarifies the distinction between the RPS class, which is the template for pipelined serial subarray hardware, and machines such as cytocomputers which are strongly influenced by specific applications. Cytocomputers are nonetheless RPS machines, but we prefer to use the more abstract class as a starting point for the study of DA engines.

3.3. Design of RPS Systems

We now proceed to analyze the basic elements of the RPS structure. The first section introduces some terminology and a partitioning of design issues in RPS systems. The second section analyzes the performance of an abstract RPS machine. The third section evaluates RPS-structured machines against the metrics for DA engines presented in Chapter II. The final section contrasts comparable RPS and array structures.

3.3.1. Local and Global Concerns in RPS Systems

RPS system design partitions into two issues: *local* issues that concern the processing of individual cells in each stage, and *global* issues that concern the movement of entire grids through the pipeline of subarray stages. Design of the hardware and software components an RPS systems divides into *local design* and *global design*. This terminology will be used throughout the subsequent sections.

Local issues pertain to subarray stage design. Referring again to Figs. 2.5 and 2.5, the three basic components of one stage are the line buffering scheme, the subarray storage, and the subarray processor. These function as follows:

- The *Buffering Scheme*, in combination with the subarray storage, aligns into subarrays the serial cell stream passing through the stage. It insures that, on successive time steps, successive subarrays of in the input grid arrive at the subarray processor.
- The *Subarray Storage* holds successive subarrays acquired from the cell stream. It is closely tied to the buffering scheme. On successive time steps, this storage holds what is effectively the *state* of the input grid.
- The *Subarray Processor* computes new cell values based on data stored in the stage. The most important data source is the subarray storage itself which represents a small piece of the input grid. This is the mechanism by which local, neighborhood computations (i.e., array-style computations) are performed. Other data sources include any previously computed data retained internally for later use. The central function of this processor is to compute a new cell value and inject it into the output cell stream for the following stage.

During each time step, one cell enters the stage, and a new, computed cell exits. This activity is one *subarray computation.* One subarray computation has three basic parts:

(i) Examine the current state of the stage, which includes the subarray storage and any internal variables.

(ii) Compute new values: a new cell to output, and any internal values to be retained.

(iii) Relocate computed values. A new cell must be injected into the output stream. Internal variables may be relocated wherever appropriate, including either the subarray storage or buffers, thus overwriting the input.

Global issues pertain to the physical properties of the pipeline, as distinct from the functional properties of a stage. Three central concerns are pipeline rate, pipeline length, and pipeline control strategy.

Despite the partition, local stage design influences global pipeline rate. Two properties of a stage are defined globally: the effective stage cycle time, and the stage latency. The stage cycle time, τ_{stage} is the time between the output of *consecutive* cells from a streaming stage. τ_{stage} depends on the algorithm executing in the stage, i.e., how many low-level operations to perform one subarray-computation, and hence also depends upon operation timing in a stage, which is a hardware design parameter. The stage latency, τ_{lat} is the time required for a *single* cell to pass completely through the stage. Latency is a function not only of τ_{stage} but also of the size of the grid being processed.

Pipeline length is of concern because it is dynamically variable. With modular, programmable stages, pipeline length can be optimized to meet cost or performance constraints. Nevertheless, as a rough metric, long pipes typically exhibit better performance than short pipes.

The principle concern in pipeline control is the implementation-dependent overhead associated with managing pipeline data movement and stage programming. As in most array systems, the existence of some global controller is assumed, the task of which is to synchronize pipeline data movement, stage programming, and the interface to the data source.

Note that the local/global partition also applies to array-structured systems, where the emphasis is usually upon local concerns. Consider a large array designed to meet bandwidth constraints: such a system will minimize loading and unloading of grids, grid edge-effects, and non-local data movement, all of which are essentially global issues. An ideal array, large enough to store an entire problem that requires strictly local computations, needs to address mostly local issues: cell computation, node design, etc. The only global problems are how to broadcast the SIMD instruction stream to all array nodes, and how to load and unload grids. Some practical arrays, on the other hand, do take special account of non-local issues. For example, MPP [Batc80] has a separate staging memory for reordering sheets of data outside the processor array; we regard this as a global concern. For RPS systems, global concerns are always important because cell stream management (IO) and subarray computation in the pipeline (processing) are concurrent activities. In contrast, most grid IO for arrays can be viewed as an unavoidable overhead at the start and close of processing.

3.3.2. Basic Performance Analysis

We analyze here the time required to process a grid of arbitrary size through an RPS pipeline of arbitrary length. Stage functionality will be ignored. It will be assumed that a stage computation takes one time step of length τ_{stage}. We examine first a single stage, and then a pipeline of stages. Latency effects, and simple cost/performance optimization are analyzed. Finally, we discuss how pipeline structure affects algorithm decomposition. This analysis generalizes and extends some earlier, narrower results pertaining to cytocomputer architecture [LoMc80].

Before analyzing a general stage it is illustrative to examine a simple example of how a grid moves through one stage. Fig 3.2 shows the state of the buffers and subarray storage on successive time steps as a small grid moves through one 3×3 stage. The movement of the grid through the stage can be visualized as a 3×3 subarray window moving across the grid as shown in the figure [LoMc80]. This illustrates the structure of a cytocomputer stage. After cell $A_{4,4}$ in the cell stream of Fig. 3.2 has entered the stage, the complete 3×3 neighborhood of cell $A_{3,3}$ is stored in the stage subarray. The subarray processor then computes the new

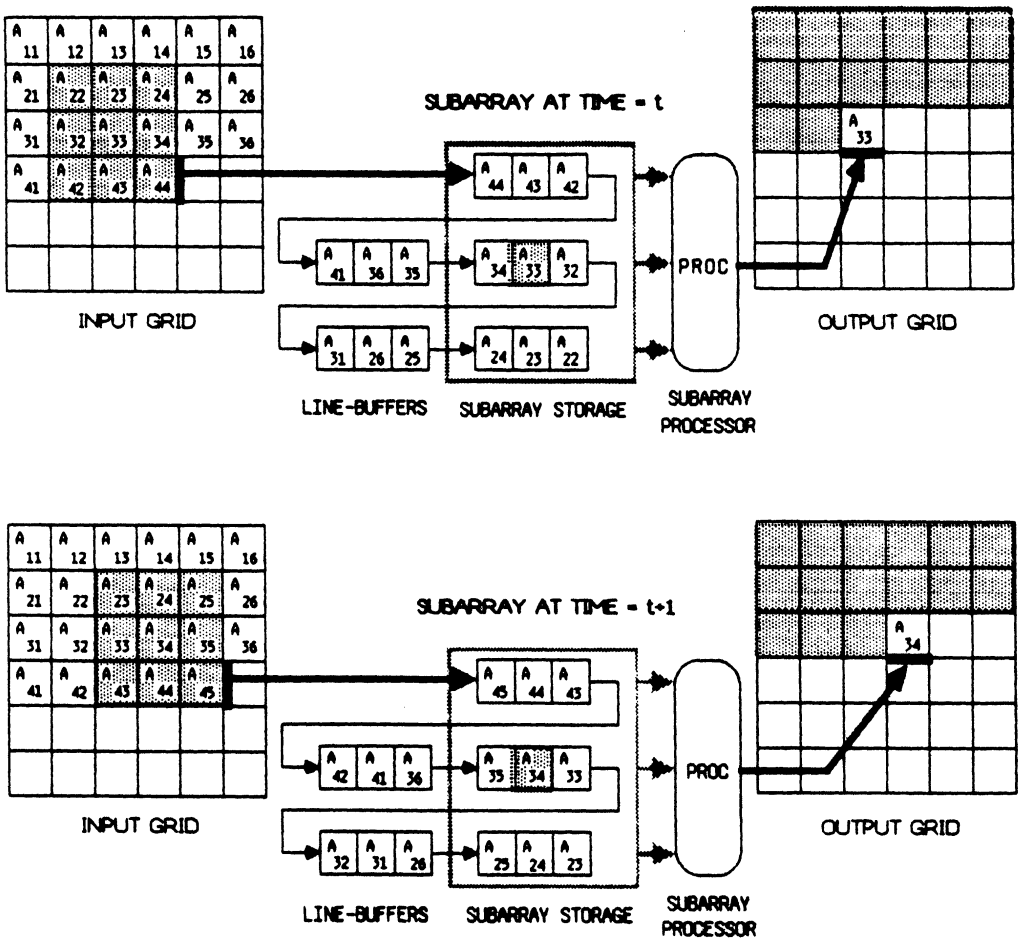


Figure 3.2 Serial Flow through a 3 × 3 Stage

value $A^{new}_{3,3}$ and injects it into the output stream. This activity requires one τ_{stage} cycle. On the next cycle, $A_{4,5}$ enters the stage and the computed value $A^{new}_{3,4}$ is output. In this example the stage latency τ_{lat} is the number of τ_{stage} cycles between $A_{i,j}$ entering and $A^{new}_{i,j}$ leaving. This is precisely the number of cells in the cell stream between $A_{3,3}$ and $A_{4,4}$ which clearly depends on the width of the input grid.

The example illustrates data movement in a subarray stage. Note that the buffers and subarray storage form one continuous shift-register through which the cell-stream flows. By tapping off values at certain points in this path-- at the cells of the subarray storage--we access a subarray in the input grid. The next, laterally displaced subarray appears at the next time step,

when this shift-register shifts.

Fig. 3.3 shows a more general stage structure. We restrict our analysis to rectangular subarrays on a square lattice. Other topologies, such as the hexagonal lattice, are not addressed here as they are rarely used in DA applications. This structure has a subarray with s_x columns and s_y rows. There are $s_y - 1$ buffers of maximum length L; to process an $X \times Y$ grid, these are configured to be of length $X - s_x$.

The first time to calculate is the stage latency τ_{lat}, which we may take to be the elapsed time between the entry of $A_{1,1}$ and the exit of cell $A^{new}_{1,1}$. To compute $A^{new}_{1,1}$, $A_{1,1}$ must appear somewhere inside the subarray storage, along with some of its neighbors in the grid. However, because it is on a corner, this cell does not possess all the neighbors of a cell in the middle of the grid. It is necessary to *define* when the subarray holds sufficient data to recompute $A_{1,1}$. Fig. 3.3 shows a distinguished cell in the subarray with (x, y) coordinates (c_x, c_y) which we term the *center* of the subarray. For the 3×3 subarray of Fig. 3.2, this was literally the center of the subarray. One subarray-computation produced a new center value as a function of the old center and eight neighbors in the subarray, and injected it into the output stream. Actually,

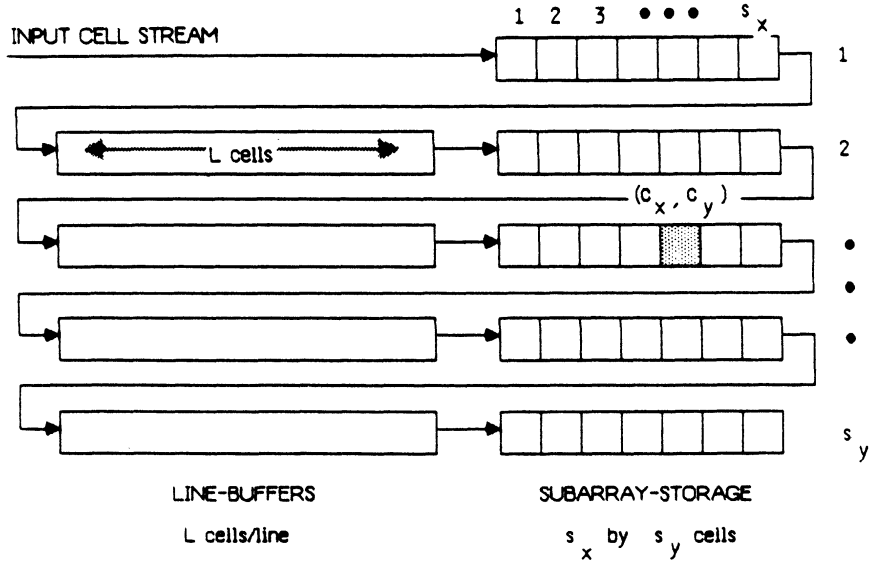


Figure 3.3 General Subarray Stage Structure

the center of a subarray is entirely arbitrary. τ_{lat} is the number of time steps necessary to shift cell $A_{1,1}$ into cell (c_x, c_y) of the subarray[3], as shown in Fig. 3.4. By inspection, the latency for a grid of size $X \times Y$ is then:

$$\tau_{lat}(X, Y) = ((c_y - 1)X + c_x) \tau_{stage} \tag{3.1}$$

This depends upon the grid width X because the total length of one line buffer plus one subarray row is precisely X. Latency is a side-effect of the need to buffer a few rows of the grid.

After τ_{lat} cycles, a new computed cell value exits the stage on every cycle. By the preceding analysis, the new computed cell occupies the same place in the output grid as the center cell of the subarray conceptually passing across the input grid. XY additional cycles then flush the remaining grid cells though the stage to complete this processing step. Define $\tau_{pass}(S, X, Y)$ to be the time required to pass an $X \times Y$ grid through an S stage pipeline. The time to pass an $X \times Y$ grid through 1 subarray stage is then:

$$\tau_{pass}(1, X, Y) = ((c_y - 1)X + c_x + XY) \tau_{stage} \tag{3.2}$$

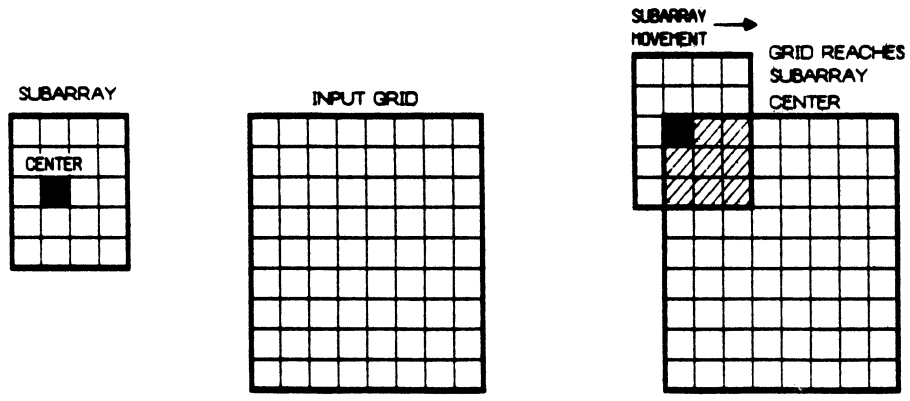


Figure 3.4 Latency to Reach Subarray Center

[3]There is a subtle side-effect of the subarray configuration of Fig 3.3. When a cell near the edge of the grid occupies the center of the subarray, there is a wrap-around effect which places cells near the *opposite* edge of the grid in the subarray. The subarray processor must distinguish that these are not valid neighbors of the center cell.

Analysis of a pipeline of stages is similar. Consider a grid of size $X \times Y$ passing through an S stage pipeline of general subarray stages. There is clearly a latency associated with the pipeline because there is a latency associated with each stage. Let τ_{lat} be the latency for one stage. Then after τ_{lat}, the first valid cell exits stage-1 and enters stage-2. After $2\tau_{lat}$, the first valid cell exits stage-2 and enters stage-3, and so on. Thus, the first valid cell exits the pipe after $S\tau_{lat}$. XY steps later, the entire grid has been processed through the pipe. Hence we have:

$$\begin{aligned} \tau_{pass}(S,X,Y) &= S\tau_{lat} + XY\tau_{stage} \\ &= S(\,(c_y - 1)X + c_x\,)\tau_{stage} + XY\tau_{stage} \end{aligned} \tag{3.3}$$

As an example, consider the 3×3 cytocomputer stage of Fig. 3.2. Substituting values

$$s_x = s_y = 3, \quad c_x = c_y = 2$$

we obtain:

$$\tau_{pass}(S,X,Y) = S(X+2)\tau_{stage} + XY\tau_{stage} \tag{3.4}$$

which is the cytocomputer processing time derived in [LoMc80]. In keeping with the analogy of a subarray window moving across the grid, [LoMc80] notes that a pipeline can be viewed as a series of 3×3 stage windows following each other across the grid, each processing the previous stage's output.

A few minor points should be addressed before proceeding. First, note τ_{stage} has been assumed to be one atomic processing step. In fact, a stage may need to execute several instructions to perform a single subarray computation. τ_{stage} is this aggregate time. For the entire pipeline to operate properly, *each* stage must have the same τ_{stage} since $1/\tau_{stage}$ is the rate at which cells flow through the pipe. It may be necessary to insert null operations to pad stages so that all share the same computation time. Second, although we speak of a pipeline of subarray stages, the RPS structure differs slightly from a typical pipeline such as a floating-point arithmetic pipe because its component stages exhibit a pipe-like latency effect. Also, this stage and pipeline latency depend upon the input grid size. Nevertheless, for a given stage algorithm and grid size, the entire RPS pipe has a setup time ($S\tau_{lat}$) and cycle time (τ_{stage}) just as an ordinary pipeline.

Latency is an important effect to consider. One simple measure of the effects of latency is the ratio of pipeline latency to total pipeline processing time. Define R_{lat} to be this fraction. From (3.1) and (3.3):

$$R_{lat}(S,X,Y) = \frac{S((c_y - 1)X + c_x)\tau_{stage}}{S((c_y - 1)X + c_x)\tau_{stage} + XY\tau_{stage}} \tag{3.5}$$

which simplifies to:

$$R_{lat}(S,X,Y) = \left[1 + \frac{XY}{S((c_y - 1)X + c_x)}\right]^{-1} \tag{3.6}$$

Latency dominates processing time for very small grids close to the subarray size. For large grids, we may assume $X \gg c_x$, $Y \gg c_y$ to obtain the approximation $R_{lat} \approx (1 + Y/S(c_y - 1))^{-1}$. The ratio of grid edge Y to pipe length S is seen to be the determining factor. Latency increases for long pipelines, decreases for large grids.

Again considering a 3×3 subarray, this approximation simplifies to $R_{lat} \approx (1 + Y/S)^{-1}$. Here, latency is less than 10% for $Y > 9S$, and less than 50% for $Y > S$.

A measure of the tradeoffs embodied in a pipeline structure is the pipeline cost/performance ratio of [Kogg81]. Specifically, we evaluate cost per calculation-rate for an RPS pipe. Cost is taken to be a linear function of pipeline length as in [Kogg81]: $\alpha S + \beta$ for arbitrary constants[4] α, β. For calculation-rate we divide the total number of subarray-computations performed on one pipeline pass by the total processing time:

$$\text{calculation-rate} = \frac{XYS}{\tau_{pass}(S,X,Y)}$$

Hence, we have

$$\frac{\text{cost}}{\text{performance}} = \frac{(\alpha S + \beta)\tau_{pass}(S,X,Y)}{XYS} \tag{3.7}$$

which expands and simplifies to:

$$\frac{\text{cost}}{\text{performance}} = \left[S\left\{\frac{\alpha}{XY}((c_y - 1) + c_x)\right\} + \left\{\alpha + \frac{\beta}{XY}((c_y - 1) + c_x)\right\} + \frac{\beta}{S} \right]\tau_{stage} \tag{3.8}$$

[4] α might reasonably depend upon the size of the stage line buffers, and thus upon c_y and X. We avoid this complication here.

The object of this exercise is to evaluate the optimum pipeline length, S_{opt}, as a function of these cost and grid-size parameters. Minimizing (3.8) gives that pipeline length with the lowest cost per subarray-computation. Differentiating and solving yields:

$$S_{opt} = \left[\frac{\beta}{\alpha} \frac{XY}{(c_y - 1)X + c_x} \right]^{1/2} \tag{3.9}$$

As a concrete example, assume a 3×3 subarray with $\alpha = 1$, $\beta = 1$. The cost here is measured in stages, and the constant overhead is comparable to a single stage. Further, assume that X and Y are large relative to the subarray. Then (3.9) and (3.6) can be approximated:

$$S_{opt} \approx \sqrt{Y}, \quad R_{lat} \approx (1 + \sqrt{Y})^{-1}$$

This suggests that a 64-stage pipe is optimal for a grid with 4096 cells/edge, and that the associated latency will be roughly 2%. Of course, this is a much simplified model of processing that does not account for the detailed structure of real applications. In particular, it does not account for the length of the algorithm to be performed. These types of optimizations will be examined relative to specific DA tasks in later chapters.

Pipeline length itself affects the decomposition of algorithms into individual steps performed in each stage. Consider an algorithm composed of many repetitions of one processing step. If this step can be realized as K subarray-computations, then a KS-stage pipe performs S of these steps on each grid pass. If the pipeline is too short, $S < K$, then $\lceil K/S \rceil$ passes are required to realize the step, and each stage performs a different function on each pass. Long pipelines are generally desirable, but, as just discussed, have a longer total latency.

3.3.3. Metrics for DA Engines Revisited

The RPS structure evolved as an alternative to array-structured hardware in picture-processing applications. One possible conclusion to be derived from this interpretation is that RPS machines are necessarily sub-optimal alternatives to arrays in the long run. That is, the availability of VLSI of wafer-scale arrays will necessarily obsolete RPS systems. We argue this is not the case, that RPS systems have several intrinsically desirable properties. This section evaluates these properties in the context of DA applications using the metrics suggested in

Chapter II.

The following discussion steps through the seven design issues for DA hardware discussed in Chapter II, summarizing the advantages and disadvantages of RPS machines relative to each. Some comparisons to array structures are also included.

- **Task Model and Architecture**

  Any task represented on a cellular grid with primarily local computation (i.e., subarray computation) is a candidate for an RPS engine, or an array. Calculations to be applied to groups of cells determine the required subarray-processor functionality.

- **Speed, Cost-Effectiveness**

  Speed comes from the ability to perform several subarray-computations concurrently in an RPS pipe. Roughly speaking, one can improve the execution speed of a task, up to a point, by increasing the pipeline length. The ability to start with a very short pipe and to improve performance by acquiring stages, as cost permits, appears to us to be particularly cost-effective: one need never acquire more hardware than that demanded by the task at hand. (Compare with a very large fixed array: the processing power is large, but so is the start-up cost.) We quantify some of these comparisons in the following section.

- **Range of Application**

  Programmable pipeline stages which perform variable subarray computations enable an RPS pipeline to address a range of problems. A completely general subarray processor, e.g., a microprocessor, is as flexible as a bit-serial array node, at the cost of some speed. A stage optimized for a restricted set of applications, e.g., maze-routing, is less flexible, likely faster, and can handle differing tasks within this limited range.

- **Task Representation, Task Limitations**

  The fundamental concern regarding task representation is how to manage large cellular grids. Note that a grid representation is often viewed as a brute-force approach since it may require flattening a more compact hierarchical representation. This applies equally well to RPS engines and arrays. We regard this as a tradeoff: compactness of representation is lost, but parallelism in processing is gained.

There are three fundamental task limitations for an RPS system. The first is locality of processing: most of the processing on the grid must be local or the task is a bad candidate for this approach. Neither RPS machines nor arrays handle non-local computation particularly well. Arrays typically accommodate this by including row/column busses so that individual nodes can be addressed. It is thus possible to insert conditional branches, based on the state of individual cells, into the SIMD instruction stream broadcast to the array by the global controller. In contrast, an RPS pipeline performs several instructions concurrently, one per stage, so arbitrary conditionals can only be evaluated after the grid exits the pipe. However, it is possible to evaluate some global properties of the grid as it streams through each stage, useful for future branches based upon global grid state.

The second limitation is cell size. An RPS pipeline must have some maximum internal datapath width, limiting the maximum size of each grid cell. Some tradeoffs may be possible, however, if the internal memory for buffers and subarray-storage is reconfigurable. Virtual arrays with large node memories can make smoother tradeoffs between grid size and cell size, and can handle small grids with very large cells.

The third limitation is grid size. RPS pipelines are attractive because, unlike arrays, the entire grid need not reside in the stage buffers. Stages with long line buffers can process very wide grids (with any number of rows) simply by streaming them through. In contrast, the entire grid must fold completely onto the node memories or it must be processed in sections. This comparison is quantified in the following section.

- **Hardware/Software Boundaries**

  In RPS systems, *hardware* includes the pipeline and anything required to store, manage, or generate the cell stream; *software* includes the host environment and the programs executing in each stage. The choice of where to implement a portion of a large task--in the pipeline or on the host--depends not only on the structure of the computation, but also on the host interface.

- **Modularity**

  A strong point for the RPS organization is that it can be built from modular, programm-

able stages. Processing capacity may be expanded incrementally. Note however, that there is a decreasing marginal return from additional stages. An S+1 stage pipe may not perform significantly better than an S stage pipe for S large[5]. Also, because the grid does not reside in the pipeline, the hardware which manages the cell stream, for example, a cell-buffer and disk, is upgradable independent of the pipeline.

In contrast, arrays have typically been built large to meet large bandwidth requirements. Modularity is less an architectural problem, more an engineering problem for arrays. There is little to be gained from adding one row/column to a large array (again, the diminishing returns effect). Signal propagation may become a problem as a small array grows larger. Highly integrated arrays [Batc80, Aoki82, GrNE84] have both memories and processors on the same chip. These arrays can be made larger, but the memory/node cannot. The practical difficulty to increase nodal memory in a tightly integrated array--widening busses, replacing chips--may be considerable. The best approach to general modularity here seems to be abutting complete, small, virtual arrays to form larger rectangular arrays [Blan84].

- **Host Environment**

  Clearly, this environment is implementation dependent. We discuss the prototype environment designed to support our experimental work later in this chapter.

3.3.4. Comparing RPS and Array Structures

A useful way to contrast RPS and array structures is to regard each as an architecture for interpreting a stream of instructions. Consider a long, linear sequence of *local* instructions. Each local instruction specifies how to recompute a grid cell as a function of its neighbors in the grid. The array broadcasts one instruction to each node and all nodes execute it in parallel. It interprets the instruction stream one instruction at a time. The RPS architecture executes a group of sequential instructions in parallel, distributing them across the pipeline one instruction per stage. The grid moves through an S-stage pipeline and emerges with S instructions

[5]Unless S+1 is a multiple of K and it is desired to perform several K-step algorithms in one pipeline pass. Then, because $S \bmod K \neq 0$, the added stage means there need be no unused stages at the end of the pipe.

performed on it. The distinction is characterized by how the two structures distribute instructions in *space* : one instruction distributed to all nodes or several instructions distributed across the pipe; and in *time* : one instruction completed at all nodes in one time step, or several instructions completed in the pipe in many time steps as the grid flows through.

To reformulate this comparison quantitatively, we introduce simple models for RPS and array structures, and for the the tasks to which they apply. These models are parameterized by several variables which completely specify a particular machine or task. Variables will be subscripted "a" for array structures and "r" for RPS structures.

Machines are specified by these six parameters:

- Processor Count N - is the total number of nodes in an array or stages in an RPS pipeline.
- Memory M - is the total memory, measured in bits, located in the nodes of an array or the line buffers of an RPS pipe.
- Datapath Width ω - applies only to RPS systems. This is the internal datapath width through the buffering, subarray, and stage processor. Cells with ω_r bits are the widest that can be directly streamed through the pipeline.
- Processor Density ρ - is the amount of non-memory hardware, measured in IC packages (chips) per processor, for one node or stage. For highly integrated processors, $\rho < 1$; for processors built of many packages, $\rho > 1$. (We avoid counting memory here, although for some real integrated processors both memory and processors reside on the same chip.)
- Processor Instruction Time τ - is the basic cycle time for a single low-level processor instruction, e.g., one bit-computation in a bit-serial array node, or one template-match in a cytocomputer stage.

The description of a task is also parameterized. It is assumed that a fixed number of local instructions are to be performed on a fixed cellular grid. Four parameters specify a task:

- Grid Edge Length E - a problem is represented on a square grid with E cells per edge.
- Grid Cell Size b - each cell in the $E \times E$ grid has b bits. The entire grid has $E^2 b$ bits.

- Processor Instructions per Operation I - is a normalizing factor allowing us to compare arrays and RPS pipelines. I_a and I_r are the number of atomic node and stage instructions respectively, to perform the same *local operation* in an array or pipeline.
- Algorithm Length K - a complete algorithm is composed of K local operations; it is an instruction stream of length K. Using I_a, I_r, we can determine how many actual processor instructions are required to implement this on an array or RPS pipeline.

Given these parameters, an RPS structure is described by the 5-tuple $(N_r, M_r, \omega_r, \rho_r, \tau_r)$. An array structure is the 4-tuple $(N_a, M_a, \rho_a, \tau_a)$. A task is the 5-tuple (E, b, K, I_r, I_a); the inclusion of I_r, I_a determines how the task runs on an RPS or array structure.

Two final parameters measure tasks and systems:

- Total Processing Time T - T_r and T_a are the total times required to run a specific task on a specific RPS or array structure, respectively.
- System Cost C - C_r and C_a are the cost in IC packages of a specific RPS or array structure. C depends upon N, ρ, and M.

One approach to comparing RPS and array structures is to fix most of these parameters, vary a few, and analyze the result. For example, a result of [LoMc80] for cytocomputers may be interpreted as follows: if $N_a = E^2$ is fixed, $I_r = I_a = 1$, and $K < N_r$, then $T_r = T_a$ implies that τ_a must decrease with increasing K. In other words, for any algorithm with fewer steps than pipeline stages, the RPS structure can satisfy a fixed bandwidth requirement T with constant stage cycle time τ_r, whereas a fixed array must decrease its cycle time τ_a with increasing algorithm length. This applies, for example, to real-time image display. One purpose of this section is to generalize this sort of analysis (i.e., remove the picture-processing constraints) to encompass more general tasks and machines. Given the number of parameters here, it is easy to formulate comparisons, but difficult to obtain insightful closed-form answers without trivializing away most of this complexity. This section examines memory utilization, processing time, and cost × time product for representative systems and tasks. Before proceeding, we need to make clear our assumptions about these RPS and array models.

The model for arrays makes the following assumptions. Arrays are always square, with $\sqrt{N_a}$ processors/edge. They are virtual arrays in the sense of [Blan82, HoNS81] where M_a/N_a bits of memory are stacked upon each node and large grids are folded onto the physical array. We assume that edge mechanisms are provided to make this transparent. If a task with an $E \times E \times b$ grid requires more memory than present in the array then the task cannot be performed. In a real system, the grid would be sectioned and processed piecewise. This introduces system-dependent effects--IO overhead, edge handling-- that are desirable to avoid in a simple model.

The model for RPS structures assumes that the pipeline has N_r stages with M_r bits of memory distributed across the line-buffers in the stages. Stages have 3×3 subarrays. Internal datapath width is fixed at ω_r bits. A task with an $E \times E \times b$ grid cannot be processed if the cells are too big or the grid itself is too wide for the line-buffers. Again, a real machine might section the grid to process it; we avoid this complication as with arrays.

The first area to examine is memory utilization. This answers quantitatively what it means for a task to be "too large." An $E \times E \times b$ grid is too large if it cannot be folded onto an array, or if its rows are too large for the buffers in an RPS stage. Consider E^{max}, the largest (square) grid edge size, as a function of N and M for an array or pipeline. Let $M = M_a = M_r$ be fixed. The grid fits onto the array just if there is sufficient memory: $M > E^2 b$. Thus, for an array:

$$E_a^{max} = \left[\frac{M}{b} \right]^{1/2} \tag{3.10}$$

and the need to convert to an integer is ignored. E_a^{max} is independent of N_a since M_a is fixed. In an RPS pipeline, we assume the memory is partitioned equally among the line buffers in N_r stages. Each stage has two buffers of ω_r-bit cells. If all memory is in the buffers, E_r^{max} is the length of each buffer, and so

$$2\omega_r E_r^{max} N_r = M$$

which yields:

$$E_r^{max} = \frac{M}{2\omega_r N_r} \tag{3.11}$$

Assume $\omega_r = b$ so that the cells can be processed in the RPS pipe. Then our basic result is:

$$E_r^{max} > E_a^{max} \text{ for } \frac{M}{b} > 4N_r^2 \tag{3.12}$$

For all but the smallest M, a practical pipeline (1-100 stages) satisfies (3.12). For example, if at least 40000 cells of memory are distributed across a 100-stage pipeline, then (3.12) holds. Hence, for a fixed memory size, the RPS pipe typically accommodates larger grids directly without sectioning.

The next areas to examine are processing time and system cost for tasks running on comparable machine structures. We regard machines as *incomparable* if comparison examines only tasks that run on one machine but not the other, or if comparison examines systems of radically different sizes. To avoid this, we only compare groups of machines. Candidate array and RPS machines are constructed by varying N, M, and ρ over reasonable ranges. Candidate tasks are constructed similarly by varying E, b, K, I_a and I_r. We evaluate the cost, C_r, C_a, of each machine and the time, T_r, T_a to execute each task on each machine. This creates a set a machine $\times$ task $\times$ performance points. Two relationships are extracted from this data: processing time as a function of problem size, and cost $\times$ time product as a function of problem size. The edge length E is taken as the problem size. By plotting these machine points we compare the relative performance over a range of RPS and array structures and tasks.

Consider a task to be executed on an RPS structure. Assume sufficient memory for the buffers. The the cost of this structure is computed as:

$$C_r = \rho_r N_r + \frac{M_r}{256 \times 1024} \tag{3.13}$$

Note we assume that memory is measured in units of 256k-bit packages. The units for C_r are IC packages (chips). The time to process this task is the time to perform KI_r subarray computations on an N_r stage pipe. There are two cases. If KI_r is a multiple of the pipeline length, then we require several passes through an N_r-stage pipe, and processing time is:

$$T_r = \left\lceil \frac{KI_r}{N_r} \right\rceil (N_r(E+2) + E^2)\tau_r \tag{3.14a}$$

If not a multiple of pipeline length, we require one extra pass with a shorter pipeline to take care of the last few instructions, and processing time becomes:

$$T_r = \left\lfloor \frac{KI_r}{N_r} \right\rfloor (N_r(E+2) + E^2)\tau_r \quad + \quad ((KI_r \ mod \ N_r)(E+2) + E^2)\tau_r \tag{3.14b}$$

The first term is the the time for pipeline passes with N_r active stages; the second term is the time for one pass through a shorter (KI_r *mod* N_r)-stage pipe for the remaining computations. It is assumed that pipeline stages can be deactivated with negligible delay.

Next consider a task to be executed on an array. Again, assume sufficient memory. As before, the cost is:

$$C_a = \rho_a N_a + \frac{M_a}{256 \times 1024} \tag{3.15}$$

where memory is measured in 256k-bit packages; C_a has units of IC packages. The time to process the task is the time to perform KI_a computations over the nodes in the array. The grid is stacked onto the array if it is larger than the physical array. If $E \leq \sqrt{N_a}$, the grid fits onto the array without stacking. If $E > \sqrt{N_a}$, the grid must be stacked or folded into $\left\lceil \frac{E}{\sqrt{N_a}} \right\rceil^2$ tiles. In either case, the processing time can be expressed as:

$$T_a = \left\lceil \frac{E}{\sqrt{N_a}} \right\rceil^2 KI_a \tau_a \tag{3.16}$$

After choosing parameters, (3.13)-(3.16) are used to calculate time and cost. The parameters for a task are varied as follows:

E	=	16, 64, 256, 1024, 4096 cells/edge
b	=	32 bits
K	=	100, 1000 local operations
I_a	=	10, 100 instructions/array-operation
I_r	=	1 instruction/RPS-operation

This generates 20 unique array tasks and 10 unique RPS tasks. We have essentially normalized with respect to the RPS structure by fixing $I_a = 1$. In addition, these parameters are fixed for both RPS and array structures:

ω_r = 32 bits/cell (same as b, width of grid cells)
M_r = 32Mbits
M_a = 32Mbits

Given that memory utilization for arrays and RPS structures has already been characterized, ω_r, M_r, and M_a are fixed at reasonable values for real systems. Table 3.1 describes the range of variation for N, ρ, and τ, and also I_r, I_a for candidate RPS and array systems. Fig. 3.5 illustrates these ranges graphically.

The choices in Table 3.1 require some justification. N, ρ, and τ are varied over ranges from "small" to "large" values. Arrays are assumed to have bit-serial nodes, so $I_a > 1$ instruction/operation, and we choose $I_a = 10$, 100 as reasonable instruction counts for 32-bit cells. Further, arrays integrate at least one node per chip, so $\rho < 1$ chip/processor; they operate with at most a 1 μs cycle time. These ranges are consistent with several real bit-serial arrays [Blan82, Batc80, Aoki82, GrNE84]. RPS structures are assumed to have fewer, larger, slower processors. However, these processors are more complex than a bit-serial node, thus we normalize to $I_r = 1$ instruction/operation for 32-bit cells. The values in the table are motivated by machines in the cytocomputer family, which have $\rho_r \approx$ 1-150 chips/stage, $\tau_r \approx 0.1\mu$s to 2μs [Loug84]. For both structures, it is assumed that larger systems attain a higher degree of integration, and so ρ decreases with increasing N.

Note that this generates 9 candidate arrays and 9 candidate RPS structures. Executing each task on each structure gives 180 array points, 90 RPS points. Partitioning by algorithm length K, we compare 90 array points against 45 RPS points for $K = 100$, 1000.

Fig 3.6 shows processing times T_r, T_a, and cost $\times$ time products $C_r T_r$, $C_a T_a$ versus problem size for algorithm length $K = 100$. T is displayed as $\log_{10}(seconds)$. CT is displayed as $\log_{10}(packages \times seconds)$. Each machine structure appears as a point on a graph. Groups of related machines are shaded: arrays with $I_a = 10$, arrays with $I_a = 100$, RPS with $I_r = 1$.

Fig. 3.7 shows the same data for $K = 1000$. Note that the corresponding regions are nearly identical, except shifted one unit higher on the T and CT axes.

This exercise reveals three important points of comparison. First, the arrays exhibit a smaller range of allowable problem sizes. As expected form the analysis of memory utilization,

Proc/Machine N	Speed (τ), Size (ρ) : 3 ranges		
	(slow) τ, ρ, I	(medium) τ, ρ, I	(fast) τ, ρ, I
(small) **4 × 4 array**	$\tau_a = 1\mu s$ $\rho_a = 1$ $I_a = 10, 100$	$\tau_a = 100ns$ $\rho_a = 1$ $I_a = 10, 100$	$\tau_a = 10ns$ $\rho_a = 1$ $I_a = 10, 100$
(medium) **32 × 32 array**	$\tau_a = 1\mu s$ $\rho_a = 0.1$ $I_a = 10, 100$	$\tau_a = 100ns$ $\rho_a = 0.1$ $I_a = 10, 100$	$\tau_a = 10ns$ $\rho_a = 0.1$ $I_a = 10, 100$
(large) **256 × 256 array**	$\tau_a = 1\mu s$ $\rho_a = 0.01$ $I_a = 10, 100$	$\tau_a = 100ns$ $\rho_a = 0.01$ $I_a = 10, 100$	$\tau_a = 10ns$ $\rho_a = 0.01$ $I_a = 10, 100$
(short) **8-stage RPS pipe**	$\tau_a = 10\mu s$ $\rho_a = 100$ $I_r = 1$	$\tau_a = 1\mu s$ $\rho_a = 100$ $I_r = 1$	$\tau_a = 100ns$ $\rho_a = 100$ $I_r = 1$
(medium) **32-stage RPS pipe**	$\tau_a = 10\mu s$ $\rho_a = 10$ $I_r = 1$	$\tau_a = 1\mu s$ $\rho_a = 10$ $I_r = 1$	$\tau_a = 100ns$ $\rho_a = 10$ $I_r = 1$
(long) **128-stage RPS pipe**	$\tau_a = 10\mu s$ $\rho_a = 1$ $I_r = 1$	$\tau_a = 1\mu s$ $\rho_a = 1$ $I_r = 1$	$\tau_a = 100ns$ $\rho_a = 1$ $I_r = 1$

Table 3.1 Candidate RPS and Array Systems

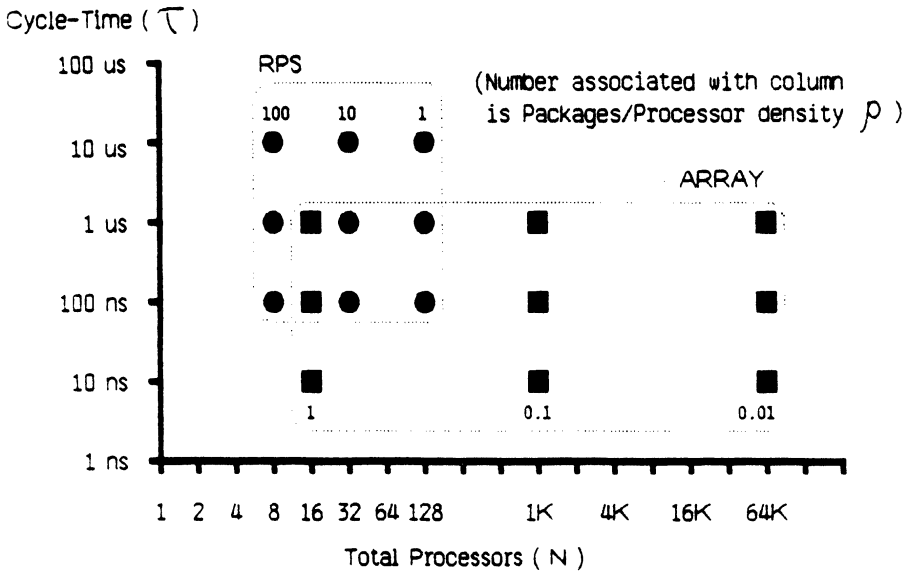


Figure 3.5 Comparison of Candidate RPS and Array Systems

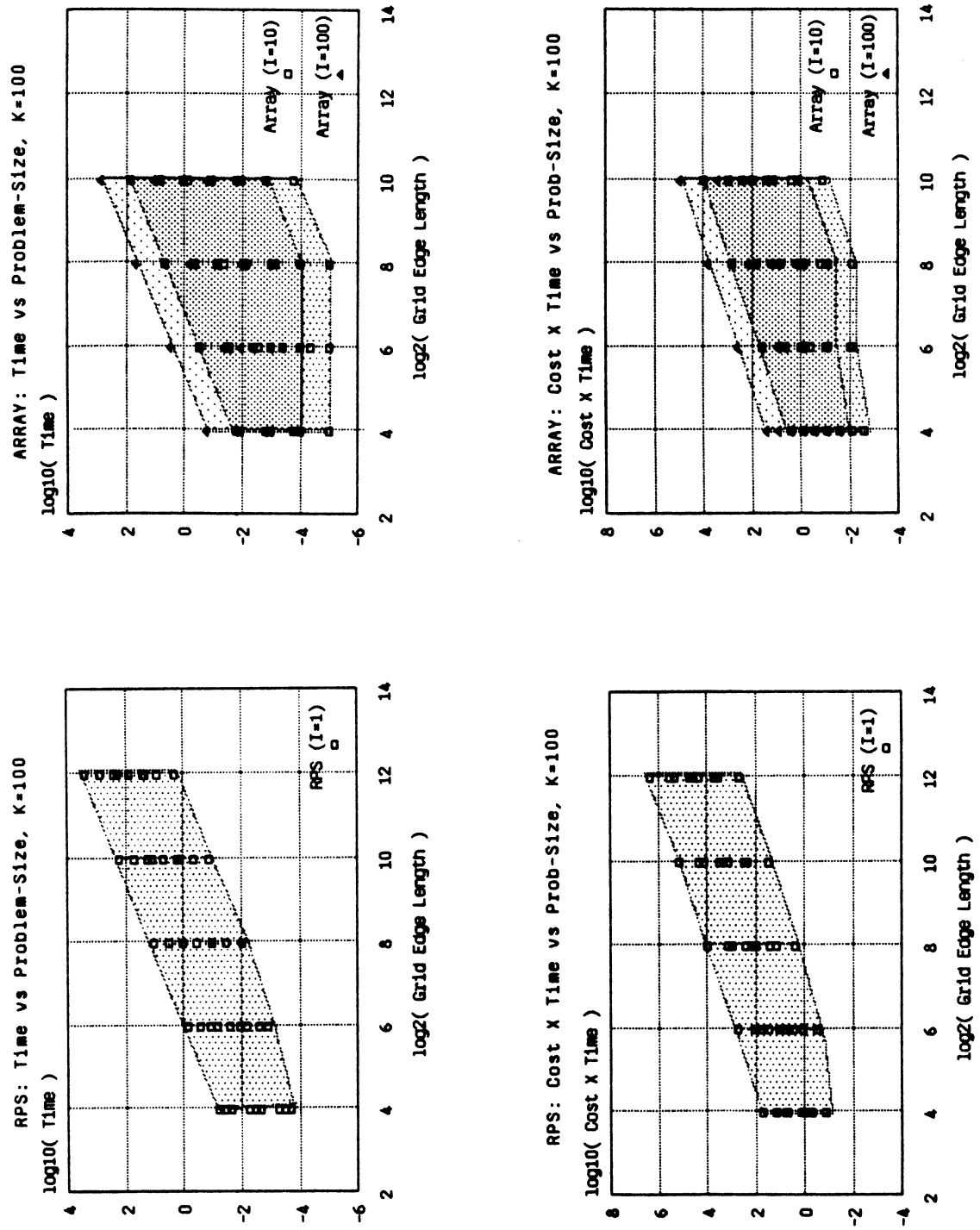


Figure 3.6 RPS/Array Time, Cost × Time vs. Problem Size: K=100

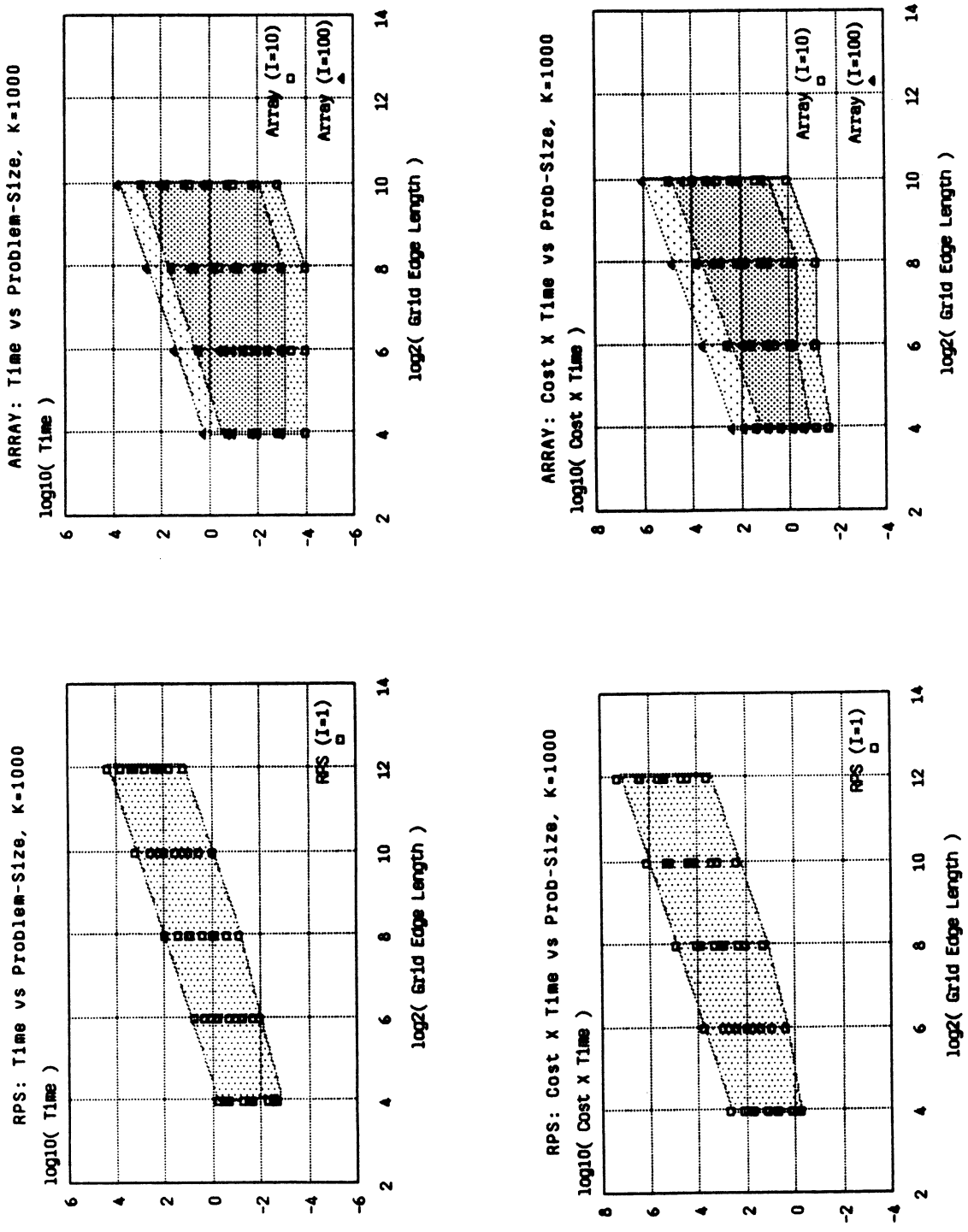


Figure 3.7 RPS/Array Time, Cost × Time vs. Problem Size: K=1000

th largest grids cannot fit without sectioning on the arrays, but can flow through the RPS pipelines. Second, the arrays occupy a broader range of T and CT--that is, the candidate arrays include faster, slower, larger, and smaller machines than the corresponding range for RPS machines. However, the third, critical observation is that there is a *common* range of achievable performance for comparable arrays and pipelines. Thus, one can choose either an array or a pipeline to meet some fixed performance or complexity constraints.

This is an important observation for the following reason. The previous section discussed qualitatively the desirable properties of RPS structures, and argued that arrays are not necessarily the superior alternative in all engineering situations. The preceding analysis suggests that these desirable properties do not degrade performance by several orders of magnitude.

Roughly speaking, the largest array systems perform better than the largest RPS systems. This results from a fundamental structural difference. Recall the instruction stream interpretation at the start of this section. We consider an array to be large if $N_s \sim 10^3$ to 10^4 processors. Such an array is desirable because it processes one instruction for a large input grid in one step; the larger the array the better. In contrast, the RPS pipeline processes several *instructions* in parallel, one per stage. Long pipes, $N_r \sim 10^2$ stages, are useful just so long as there are algorithms of corresponding length. We shall show later that long pipes are useful in some DA applications. Pipelines with 10^3 stages or more appear to be less useful for DA applications. Finally, despite the fact that this section constructs metrics to analyze fixed tasks and fixed structures, the ability to reconfigure a pipeline dynamically is of central importance in practical applications. The cost and performance of real pipelines will change with pipeline length.

3.4. An RPS Environment for DA Studies

This section describes an RPS environment based on a commercial minicomputer and commercial RPS hardware, integrated to support experiments on DA tasks.

3.4.1. Hardware and Software

Fig. 3.8 shows the global configuration of the system. The host is a DEC VAX 11/780[†], connected to an ERIM Cytocomputer II[††]; the host software is constructed as several layers running in UNIX 4.1bsd[†††]. The hardware is described first, followed by the software.

The RPS hardware consists of a cell-buffer, a global controller, and a pipeline of subarray stages, with properties summarized in Table 3.2. Cytocomputers exist in MSI and LSI implementations. Custom LSI and semi-custom stages have been fabricated with τ_{stage} between 100ns and 2μs. The hardware used in this system is a prototype for subsequent models.

Each stage has a 3 $\times$ 3 subarray to process 8-bit cells, and performs the following steps in one τ_{stage} cycle (see [LoMS80] for details):

Step-1: A cell enters the stage, is biased (normalized) or has some of its bits masked.

Step-2: The nine cells of the stored subarray are transformed into a nine-bit vector. Each bit is a true/false decision about each cell, the result of a threshold comparison with an arbitrary constant. This vector is an address into a table.

Step-3: A new cell value is selected from the following: the old center value in the subarray, the largest value in the subarray, or the value in the table addressed by the nine-bit address from (2).

Step-4: The cell from (3) is unbiased and unmasked, i.e., step (1) is selectively undone. It is possible to alter only a single bit in this cell as a function of all the bits of the subarray.

Step-5: The cell from (4) addresses another table to produce a modified cell value. This table is used typically for Boolean operations on bits or for further arithmetic biasing.

Step-6: The cell from (5) is injected into the output stream. It occupies the same location in the output grid as the center cell of the current subarray.

Consistent with the model of one state-transition, the computed value of Step-6 is never retained internally for further computation. Neither is there any extra storage for temporary or global data. This property of the cytocomputer stage will be referred to as *statelessness*. This stage architecture, and statelessness specifically, reflect design choices motivated by the image-operators of Sec. 3.2.2. The bias-values, masks, constants, and tables in Steps 1-6 are the instructions for a single stage, called a *stage program*. Because the stage depends heavily upon table look-up the perceived format of the bits in each cell is arbitrary. Unlike the general RPS

[†]VAX is a trademark of the Digital Equipment Corporation.

[††]Cytocomputer is a trademark of the Environmental Research Institute of Michigan.

[†††]UNIX is a trademark of AT&T Bell Laboratories.

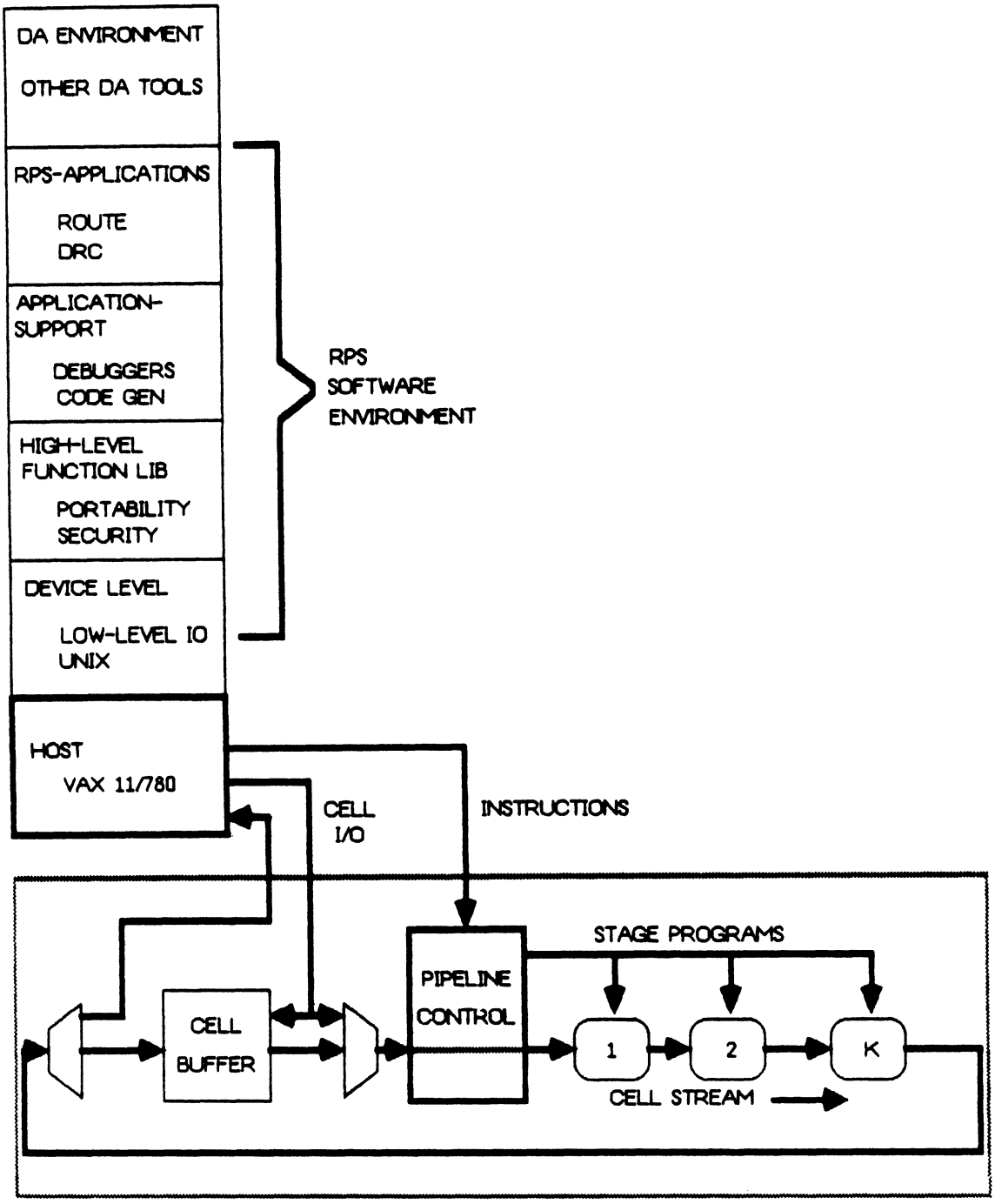


Figure 3.8 RPS System Configuration

Component	Description
RPS Hardware	Cytocomputer II
Subarray Stage	3 × 3, limited arithmetic extensive table-lookup
Cells, Line-buffers	8-bits/cell, 4096 cells/line
Cell-buffer	256k-cells
τ_{stage}	$2\mu s$
Pipeline Length	Variable
Pipeline Controller	Microcoded, with low-level instruction set
Host	VAX 11/780
Host-RPS Connection	16-bit DMA channel
Host IO-overhead	~ 15-30ms/RPS-command

Table 3.2 RPS System Hardware

stages discussed up to now, each stage-computation on this hardware is one atomic step. Despite the general template-match architecture (Steps 2,3) the table mechanisms exhibit sufficient side-effects to accomplish some of the computations necessary for DA problems.

The system is configured as a host with an attached cytocomputer. The host sends instructions to the global controller, a microprogrammed unit that programs the stages and manages the pipeline. The controller provides a low-level instruction set (that is, one not dedicated to a specific application) to process grids through the pipeline. In a typical processing step, the host deposits a grid in the cytocomputer cell-buffer, instructs the controller to send it repeatedly through the pipeline and back into the buffer, then retrieves some portion of the processed grid from the buffer.

The ideal pipeline processing time $\tau_{pass}(S,X,Y)$ for 3 × 3 stages given in equation (3.4) is affected by the system configuration. The actual time to process an $X \times Y$ grid K times through an S-stage pipe (i.e., KS computation steps) is at worst:

$$\tau_{sys} \approx \tau_{pi} + S\tau_{si} + K\tau_{pass}(S,X,Y) \tag{3.17}$$

$K\tau_{pass}$ is the basic time for K pipeline passes. The other terms are lumped delays: τ_{pi} the time to initialize the pipeline and $S\tau_{si}$ the time to set up all S stages. These terms arise because of the non-negligible time to dispatch low-level instructions from the host to the controller. Although small, these delays are always larger than τ_{stage}, e.g., 15ms versus $2\mu s$ on our

hardware, yielding an effective cost of several thousand subarray-computations for each controller instruction.

Users interact with the RPS hardware through layers of software running on the host. Earlier cytocomputer environments have been dedicated to picture-processing, and based upon an interpreter running image-operators in the pipeline. These sufficed for early feasibility studies [MuLT81a, MuLT81b, MRLA82, RuMA83a] but proved neither flexible enough nor fast enough for large-scale DA applications. The new environment is designed is to make DA application development appear to be ordinary software development by hiding details of the RPS hardware. Host software is divided into five layers. All software is written in C. The layers have the following functions, from bottom to top:

- **Device Layer**

  This layer provides the physical connection from host to pipeline. Command dispatch, hardware interrogation, grid IO, pipeline processing, and synchronization with the host are provided at this level. These device drivers fit into the UNIX kernel to provide standard logical entry points (read, write, etc.) to the physical RPS hardware.

- **Function Library Layer**

  This layer provides high-level entry points to the hardware and has three purposes. First, it insulates the higher level application layers from the hardware, thus allowing them to be debugged concurrently with lower levels. Second, it encourages portability: large applications which access the hardware only through these entry points can be moved to other hosts if the device and library layers can be ported. Third, it makes application development easier and safer by providing complex functions not actually available as primitives in the RPS hardware, and by insuring that dangerous system functions are inaccessible to the average user.

- **Application Support Layer**

  In this layer reside software tools to support development of RPS application codes. We have implemented an interpreter to allow interactive debugging of stage programs running in the pipeline. There is also software to support construction and analysis of stage pro-

grams. These tools suppress enough details of the pipeline hardware to make it possible to regard application development as ordinary software development.

- **DA Application Layer**

  This is where user codes such as routers, rule-checkers, etc., are written. An RPS-based tool must mesh with the other tools in the overall DA environment, and with the actual pipeline hardware. An application consists of host programs and stage programs.

- **DA Environment Layer**

  DA tools necessarily talk to each other. This environment is simply the aggregate set of DA tools into which a specific RPS-based tool must fit. Without this fit, the RPS hardware is useless, since it cannot acquire tasks built by other tools or pass on its results.

The lowest three levels are essentially fixed, and are implemented as approximately 4700 lines of C. DA application and environment codes vary. As an example, software for one-layer, two-layer, and global routing experiments totals about 13,500 lines of C for these higher layers. Because this thesis concentrates largely on maze-routing there has been extensive software development to support this task.

3.4.2. RPS Application Design

We sketch here the steps required to develop DA applications in the prototype RPS environment. This is the methodology followed in the design of the experiments presented in the following chapters. There are three steps:

- **Partition tasks into local and global sub-tasks**

  Roughly speaking, the pipeline performs local grid computations, and the host does most else.

- **Design and debug local algorithm**

  Local computations are implemented as a series of stage programs in the pipeline. Design involves the choice of the grid cell encoding (loosely analogous to a data structure), and the mapping of the necessary computations onto the available stage architecture. One apparently atomic calculation may require several stages. The *local algorithm* is physically

a C program which emits all the necessary raw stage programs. These can be debugged in isolation using the interpreter; they can be interactively executed in the pipeline.

- **Design and debug global algorithm**

  The global algorithm is responsible for grid IO, pipeline processing, and anything that cannot be mapped into local stage programs. Because cytocomputer stages are stateless, and because the pipeline controller provides only limited functions, all global data movement and global computations are done on the host. This global algorithm is implemented as a set of C programs that access the pipeline through the lower level layers. Much global design is concerned with optimizing grid IO, pipeline reprogramming, etc, to minimize the communication overhead between the host and the pipeline. This code, like any ordinary piece of software, is debugged by running it. Lower level software insures that errors or program aborts are handled gracefully in the RPS hardware. In addition, these levels provide some simple tracing functions.

3.5. Summary

This chapter analyzes RPS architecture in detail. We show that the class derives from the notion of pipelined serial subarray computation and serves as a template for hardware with this structure. Antecedents to this idea are surveyed. The RPS class is the appropriate level from which to proceed toward DA applications.

Some terminology for describing RPS systems is introduced, and basic performance issues are analyzed. It is shown that RPS structures have several desirable engineering properties related to DA applications. Further it is shown that performance is not seriously compromised, relative to comparable array structures, to attain these features. A prototype hardware/software environment for RPS DA experiments is outlined. With this background, subsequent chapters proceed to study DA problems in this environment.

CHAPTER IV

ROUTING IN AN RPS ENVIRONMENT

4.1. Introduction

This chapter examines maze-routing in an RPS environment. We first review maze-routing in context with other routing techniques, then introduce the basic mapping of a maze-routing algorithm onto an RPS pipeline, and sketch the central issues to address in order to construct functional routers. We then construct a sequence of increasingly complex routers for our prototype RPS environment: a one-layer router, a two-layer router, and a global router. Local design, global design, and experimental performance measurements are described for these systems. Generic optimization issues uncovered in these experiments are analyzed. Finally, we examine a few interesting connections to other work.

4.2. Maze-Routing Reviewed

Maze-routing, as originated by Lee in [Lee61], is a technique to find a path between points in a cellular grid optimal with respect to some cost metric[1]. These cost metrics are called path-functions. The most common function is simply path-length, and hence the algorithm can be used to find shortest paths.

We illustrate basic maze-routing using the shortest path metric. There are three phases in the algorithm: *wavefront-expansion*, *backtrace*, and *cleanup*. Given a grid whose cells are labeled as *free* or *blocked*, the goal is to find a shortest path from a *source* cell to a *target* cell, where a

[1]Lee's work showed how to find a path in a cellular grid, and is based on earlier work by Moore [Moor59] that finds a shortest path between vertices in a graph, which Moore terms a maze. Routers based on this idea are often called Lee or Lee-Moore routers.

path may cross a free cell, but not a blocked one. The pair of source and target points is called a *net*, or a *from-to*; in general a net may contain any number of points. In simple form, maze-routing works as follows (see Fig. 4.1):

- **Wavefront Expansion**

  This phase effectively labels a grid cell at location (x, y) with the length of a shortest path from the source to that cell. Labeling proceeds in a sequence of wavefronts expanding from the source cell. Cells on each wavefront are equidistant from the source. This produces the characteristic diamond-shaped wavefront in Fig. 4.1. At each step, the set of labeled cells with neighbors not yet labeled is the wavefront. These cells and their neighbors are *active* in the sense that computations are performed on them. Wavefront neighbor cells can be labeled with the length of a shortest cell-to-source path because they are adjacent to previously labeled cells, i.e., this requires only local computation. Neighbors thus labeled are added to the wavefront; cells all of whose neighbors have been labeled are said to be *expanded* and are removed from the active wavefront. One step in this process can be viewed as breadth-first search of depth one in the area adjacent to the labeled region. The order in which cells are visited here is called the *expansion-order*. This phase terminates when the target cell has been expanded (success) or when no more cells an be labeled (failure).

- **Backtrace**

  This phase traces an optimal path from target to source through the expanded cells. In practice, it is unnecessary to label cells with the values of the path-function (distance-to-source in Fig. 4.1). We need only record in each cell sufficient information to find a path back and relabel it as such.

- **Cleanup**

  After a path has been traced, the final phase is to remove from the grid all extraneous labels placed during the wavefront expansion phase, and relabel the path found as a new obstacle.

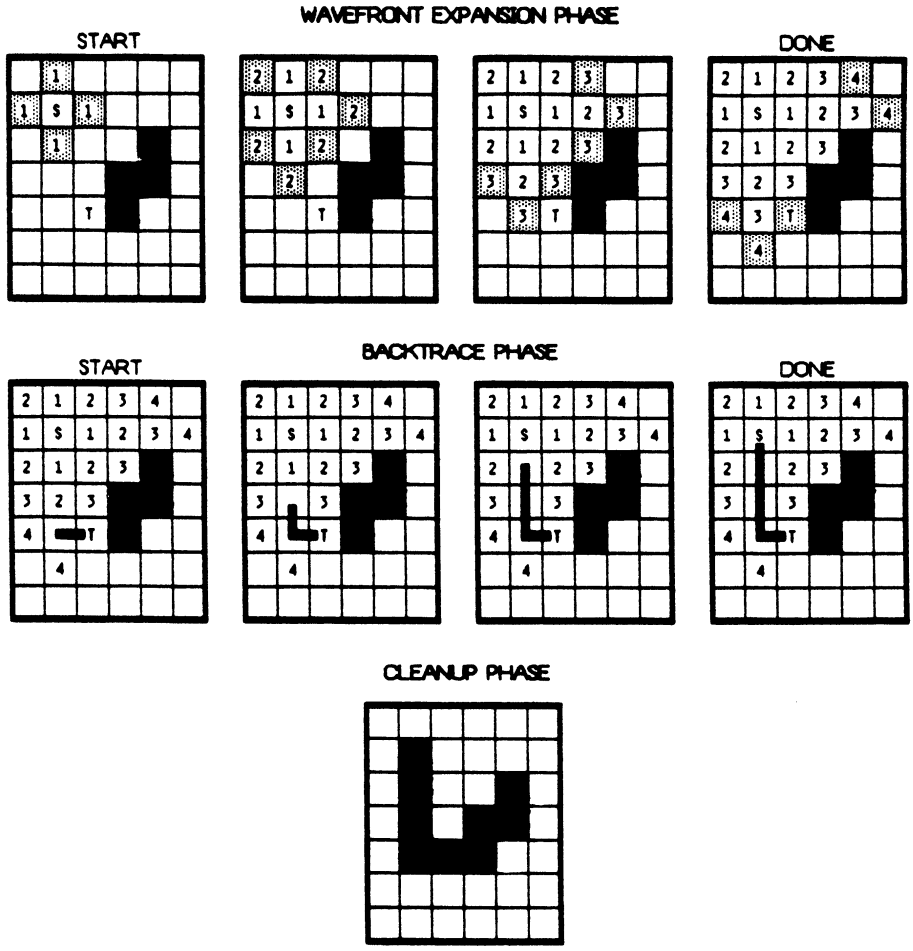


Figure 4.1 Maze-Routing Phases

This basic algorithm has been subject to substantial elaboration and refinement (see [Souk81, High83] for surveys). Research has focused upon the wavefront expansion phase because this phase consumes the most CPU resources: execution time is proportional to the number of cells expanded to find each path; memory storage size depends upon the number of bits required to encode each cell. [Aker67] originated a widely used cell-encoding requiring only 2-bits/cell, applicable when the path-function is shortest distance. Most work, however, deals with algorithms and data-structures for the expansion-order problem. [Hoel76] examines efficient list structures to manage the active wavefront cells and the search for cells to expand. [Rubi74] presents a depth-first cell expansion algorithm which expands preferentially toward the

target, thus visiting fewer cells and requiring less execution time. [Korn82] presents an implementation based on preferential expansion. [Souk78] discusses speedups obtained by controlling the expansion-order. [JiKo81] expands independently in the four quadrants surrounding the source cell. [Aker72, TYKS80] impose an arbitrary frame around a net to be routed, and expand only within the frame, thus limiting the admissible cells to be searched.

Maze-routing has two basic advantages over other routing techniques. First, it usually guarantees that a source-to-target path will be found if one exists. Second, paths can be optimized with respect to complex objective functions, for example, via-consumption and electrical cross-talk can be minimized, cross-over and pin-blocking can be avoided. This flexibility enables maze-routers to be used in a variety of routing applications and technologies. These two advantages, however, are costly, and the maze-routing approach has four main disadvantages. First, maze-routing requires large amounts of storage because of the grid representation. Complex path-functions which require complex cells only exacerbate the problem. Second, because of cell-by-cell expansion, maze-routers have long execution times. Third, despite guaranteed path-finding ability, maze-routers typically consider only one net at a time. There are no guarantees in the topological sense that all nets in a problem can be routed: paths found early may block later paths. Moreover, the fourth disadvantage is that the approach admits less formal, topological analysis of routability than other approaches.

Because of these disadvantages, maze-routing has been somewhat supplanted in recent years by other techniques, notably line-probe routers for printed circuit board (PCB) problems, and channel routers for LSI/VLSI problems. Maze-routers have not disappeared, however. Commercial maze-routers are widely available for multi-layer carriers. Because of their path finding ability, they are widely used as overflow routers to embed those final nets unroutable by other means. They are also used as global (loose) routers, which assign nets loosely to routing regions (channels) but not to detailed wiring tracks; see [SoRo81, NHLV82].

The disadvantages of maze-routers are not necessarily unresolvable. [High83] argues persuasively that the storage and time penalties associated with the approach are largely mitigated by modern virtual memory machines. The longer execution times can be viewed as the cost of the added flexibility. Intelligent use of penalties in the grid, and a path-function that finds least

costly paths, e.g. [High83, NHLV82, Korn82] reduces the one-path-at-a-time congestion problem. Moreover, maze-routing has been the exclusive technique embodied in DA routing engines, where the inherent parallelism of the expansion phase can be exploited to reduce execution time. This chapter focuses on how we might retain the advantages of maze-routing, and reduce the time burden by using RPS hardware.

4.3. Elementary RPS Maze-Routing

This section outlines how a maze-routing algorithm can be mapped onto RPS hardware, and presents some elementary performance comparisons for software, array, and RPS maze-router implementations.

Consider again the three phases of a maze-router: wavefront expansion, backtrace, and cleanup. Wavefront expansion is an inherently parallel cellular task. Cells are labeled based only on the status of neighboring cells, and all active cells can be labeled in parallel. Wavefront expansion is thus a candidate for RPS hardware. Fig. 4.2 illustrates how wavefront expansion is implemented in one subarray stage. We assume that the computation required to label a cell can be realized as one subarray computation. As the grid flows through the stage, each cell with neighbors on a wavefront is labeled as being on the wavefront. We call this one *wavefront*

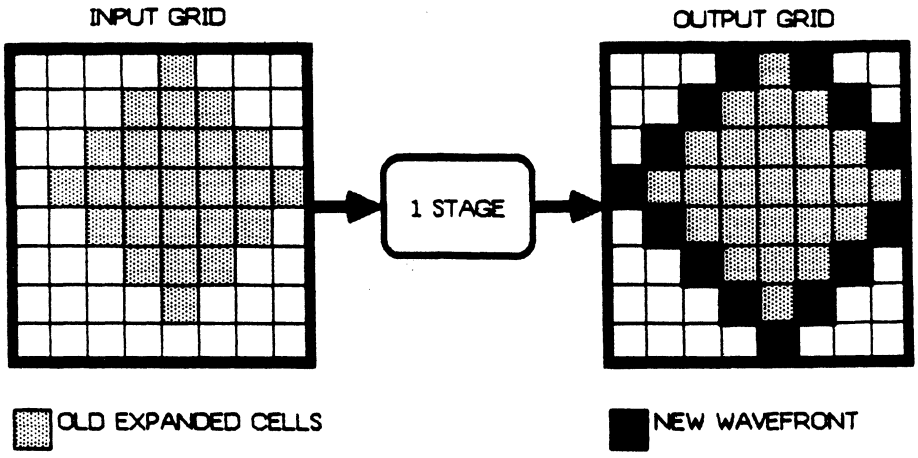


Figure 4.2 Wavefront Expansion in One Subarray Stage

expand step: it is the addition of one layer of cells to an existing wavefront. The grid emerges with the wavefront expanded one cell in each direction. In terms of the internal mechanics of a stage, we relocate a newly computed cell by injecting it onto the output, and do not overwrite the subarray storage or buffers. In this we again emulate a space of state machines. However, unlike state machines, there are several useful global computations that can be performed as the grid flows by. For example, it is useful to count how many cells are expanded, and to flag when a target cell is expanded.

Although simple, the implementation of wavefront expansion by a single-stage RPS pipeline is unappealing for the reason that *all* grid cells must be examined, even though only wavefront neighbors are being labeled. Compared to software approaches which maintain only a list of active wavefront cells at each expansion step, the hardware approach seems quite inefficient. Brute-force, in the form of fast stage hardware, mitigates this problem somewhat, but a single stage exploits none of the parallelism of wavefront expansion.

An RPS pipeline can expand cells in parallel as shown in Fig. 4.3. Each stage operates on a different version of the grid, and expands a different wavefront. That is, stage k has within

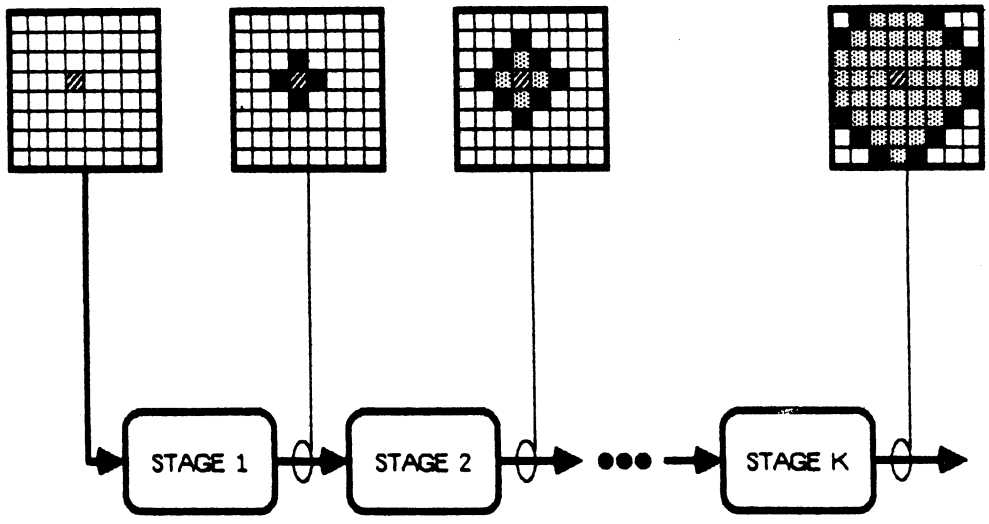


Figure 4.3 Wavefront Expansion in an RPS Pipeline

its buffers a few rows of the grid on which $(k-1)$ wavefront expands have been performed. Stage k emits a grid on which k expands have been done. Note that the kth version of the grid never exists in one place at one time. An S-stage pipeline adds S layers of cells to the active wavefront, performing S wavefront-expands in our terminology. In practice, it is the combination of stage speed and pipeline length that exhibits a potential speedup of the wavefront expansion phase.

The backtrace phase proceeds when a target cell has been expanded. Unlike wavefront expansion, backtrace is entirely sequential: exactly one cell on the path is traced at each step. Backtrace still requires local grid computations to determine the direction of the trace to the next cell. It is possible to implement this in the pipeline, but it is extremely inefficient: each stage relabels exactly one cell, and ignores the rest. Backtrace is best handled by an ordinary serial processor such as the host.

The final cleanup phase removes extraneous cell labels and marks the traced path appropriately so that further nets can be routed. Cleanup is clearly a parallel activity since all cells can be labeled concurrently. It might appear that cleanup is actually a scalar activity in which cells are relabeled based solely upon their own value, and not upon their neighbors. This is not the case if the traced path is to influence subsequent nets at a distance. Regions adjacent to vias or bends in the path may be labeled with increased penalties, so that future nets preferentially avoid these areas to minimize congestion or crosstalk. Hence, cleanup, like wavefront expansion, also requires local processing. Unlike expansion, it is independent of path length, and will generally require only a few subarrary computations (i.e., a short pipeline).

It is illustrative to compare the basic performance of software, array, and RPS implementations of maze-routers. Because wavefront expansion dominates execution time, consider only this phase. Fig. 4.4 gives a simple problem: search for a path of length L in the grid shown. The shaded area shown includes all cells within a Manhattan distance of L; the maximum wavefront is at a Manhattan *radius* of L. In the worst case, all cells within Manhattan radius L from the source will be expanded.

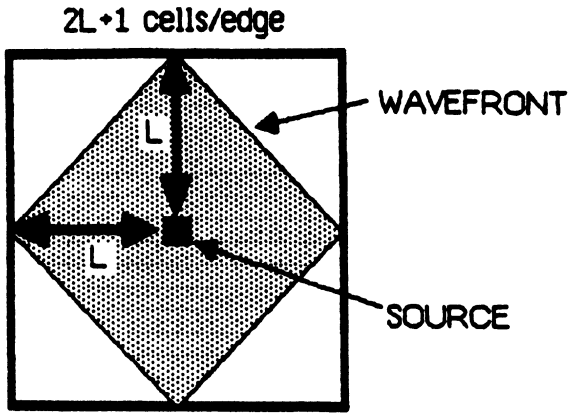


Figure 4.4 Wavefront Expansion Problem for Comparing Maze-Routers

An ordinary software implementation will at worst visit each cell in the shaded area. Its time complexity is the number of cells expanded here:

$$T_{software} = 2L^2 + 2L + 1 = O(L^2) \tag{4.1}$$

It should be noted that we have tacitly assumed a shortest distance path-function which never revisits labeled cells. More general path-functions may revisit labeled cells, because the shortest path, which is found first, is not necessarily the least costly. This can increase complexity to as much as $O(L^3)$ [NHLV82], but ordinarily few cells are revisited. Next, consider a square array structure with P processors. Assume $L^2 > P$, so that the grid must be partitioned into separate tiles and folded onto the array. Wavefront expansions must be performed on each tile, so from (3.16) the time complexity is:

$$T_{array} = \left\lceil \frac{2L+1}{\sqrt{P}} \right\rceil^2 L = O\left(\frac{L^3}{P}\right) \tag{4.2}$$

Finally, consider an S-stage RPS pipe where $L > S$. Assume one stage has a 3×3 subarray and suffices for one wavefront expand step. Then one pipeline pass performs S wavefront expands, and approximately L/S passes are needed in all. Using (3.14a) this gives time complexity:

$$T_{RPS} = \left\lceil \frac{L}{S} \right\rceil \left[S(2L + 1 + 2) + (2L + 1)^2 \right] = O\left(\frac{L^3}{S} \right) \quad (4.3)$$

These results do not immediately suggest any speedup: although time decreases with additional processors, the comparison with software yields speedups of $O(S/L)$, $O(P/L)$, for RPS and array structures, respectively. A design goal for array structures is to have the area of expansion fit onto the array with only a few tiles, i.e., $L^2 \approx P$, which reduces T_{array} to $O(L)$. Ideally, the array is as big as the entire routing grid (as with the machines of [BrSh81, Iosu80]), so that each wavefront expand step takes O(1) time steps. Design goals for an ideal RPS system are less obvious. However, if $L \approx S$, then $O(1)$ pipeline passes perform the necessary expansions in $O(L^2)$ time steps. In practice, the potential speedup for an RPS structure, compared to an $O(L^2)$ software implementation, arises from two sources. First, we assume special stage hardware to expand cells rapidly. Second, there is no overhead associated with managing lists of active wavefront cells: the grid is accessed rapidly in simple raster order, injected into the pipe, and emitted with all the appropriate cells labeled.

Software implementations exhibit no parallelism. At each expansion step, at worst $O(L)$ wavefront cells are expanded. Array implementations trade space complexity (processors) for speed. However, only cells adjacent to the wavefront are actively "processed"; most grid cells are inactive and are ignored. [Adsh82a] cites an average of only 60 cells labeled in each wavefront expand step on the 4096 processor DAP array router. RPS hardware is less parallel than an array: S stages can expand S cells concurrently, but all cells, even inactive cells, must be examined as they stream through the pipe. Because cells flow through the pipeline at a constant rate, the subarray-computation that ignores an inactive cell takes as much time as the computation that expands an active cell.

The design of practical RPS routers raises several questions in light of these comparisons. These include system-level effects such as communication overhead in the host environment, the $O(L^2)$ performance problems for long nets or large grids, and the effects of varying pipeline length. To address these questions we design and analyze three different RPS routers.

4.4. One-Layer Routing

We first construct a router to connect multi-point nets in a single interconnect layer. This router functions in the experimental RPS environment described in Chapter III. Local design is addressed first, followed by global design. Performance characteristics are deduced from measurements of single-net routing experiments. The RPS router is then compared with a software maze-router for a PCB problem.

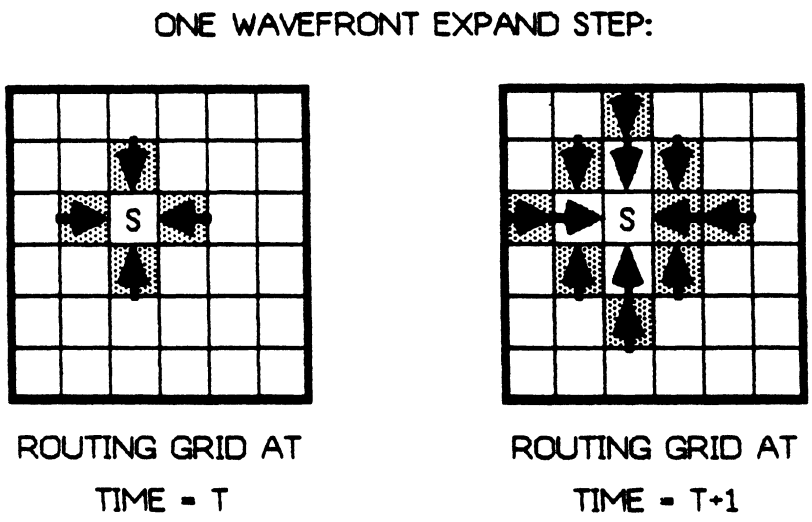
4.4.1. Local Design

Local design is concerned with local computations on grid cells. The limited arithmetic capabilities and cell size of cytocomputer subarray stages preclude the use of any path-functions that require grids with weighted cells; it is impossible to manipulate cell costs or penalties in these stages. Hence, we construct a unit-cost maze-router (crossing any cell incurs the same cost) to find paths of minimum length. Wavefront expansion, backtrace, and cleanup are considered in turn.

Several minimal-size cell encodings are possible for wavefront expansions. The basic requirement is that each wavefront be distinguishable from the preceding and succeeding wavefronts, and that sufficient information is present in the grid to trace a path. We choose a modified version of the encoding used for the L-Machine router [BrSh81]. Cells are encoded with the following alphabet:

$$\Lambda_{1\text{layer}} = \{\ source,\ target,\ free,\ blocked,\ \uparrow,\ \downarrow,\ \leftarrow,\ \rightarrow,\ trace\uparrow,\ trace\downarrow,\ trace\leftarrow,\ trace\rightarrow\ \}$$

To route a 2-point net, we find a path over *free* cells from a *source* cell to a *target* cell. Wavefronts are marked with source-pointing arrows. Each cell on a wavefront is equidistant from the source and begins a unique path to the source. Wavefront expansion is illustrated in Fig. 4.5. Expansion proceeds until a target is relabeled with an arrow, at which time there is a unique shortest path along the arrows to the source. To describe the necessary subarray-computations, we refer to cells in the 3×3 subarray by compass directions. Although implemented as an atomic template-match in the subarray stage, expansion can be expressed algorithmically as in Fig. 4.5. A cell is added to a wavefront (given an arrow label) if it has an



Given: a 3×3 subarray in the routing grid.

Task: expand subarray center cell.

Algorithm: Wavefront_Expand

```
begin
        active_labels := { ↑, ↓, ←, →, source }
        if center = free
        then begin
                if north in active_labels
                then center := ↑
                else if west in active_labels
                then center := ←
                else if east in active_labels
                then center := →
                else if south in active_labels
                then center := →
        end
end
```

Figure 4.5 One-Layer Local Wavefront Expansion Algorithm

active *free* neighbor marked as an arrow or a *source*. The arbitrary order of tests in the algorithm resolves path conflicts: several shortest paths may exist between two points, but we elect the unique one determined by the ordering.

This design also suffices for multi-point nets. Using the conventional approach of [Aker72], one net point is a *source* and the rest are *targets*. After wavefront expansion reaches at least one target, each target-to-source path is traced. The shortest path is selected, and *each*

cell on it is relabeled as a new source. Wavefront expansion proceeds again using the entire path as a source. This continues until all connections in the net are made. In this manner Steiner-points--connections in the middle of wire segments--are constructed.

Backtrace follows the source-pointing arrows from a target to a source. Backtrace can be performed in the pipeline, with each stage tracing exactly one cell as the grid goes by. This is the approach taken in our early feasibility studies [MRLA82, RuMA83a] and is extremely inefficient due to the entirely sequential nature of the processing[2]. A better approach is to perform this operation on the host. The algorithm appears in Fig. 4.6. Backtrace adds a *trace* label

---

Given: a *target* cell marked with an arrow.
Task: trace path to source, marking cells; compute path length.
Algorithm: Backtrace

```
begin
        cell := target
        path_length := 0
        repeat
                if cell = ↑ then begin
                        cell := trace↑
                        cell := north(cell)
                end
                else if cell = ← then begin
                        cell := trace←
                        cell := west(cell)
                end
                else if cell = → then begin
                        cell := trace→
                        cell := east(cell)
                end
                else if cell = ↓ then begin
                        cell := trace↓
                        cell := south(cell)
                end
                path_length := path_length + 1
        until cell = source
end
```

Figure 4.6 One-Layer Local Backtrace Algorithm

---

[2]Early studies adopted this approach only because of the inability to remove the grid from the interpreted image-operator cytocomputer environment for host processing. The new RPS environment of Chapter III was designed to address these sorts of difficulties.

to each cell on the trace path, and computes the path length. That is, wavefront labels ↑, ↓, ←, →, become trace labels *trace*↑, *trace*↓, *trace*←, *trace*→. The algorithm shown suffices for 2-point nets. For multi-point nets, we first suppress the relabeling of cells and simply compute the path length for each target-to-source path. Then the shortest path is marked with *trace*.

Cleanup removes extraneous labels from the grid, setting it up for the next net to route. With this encoding scheme, cleanup is actually a scalar operation; each cell is recomputed from its own value and not its neighbors. This fits well with the table-lookup mechanisms of the current hardware. The only complication here is this need for two separate cleanup routines: final-cleanup, in which a traced net becomes an obstacle for future nets, and intermediate-cleanup, in which a traced segment of a multi-point net becomes a *source* for future expansions on the same net. Fig. 4.7 shows the algorithm based on simple table-lookup for each cell:

The encodings and algorithms chosen here have the virtue of being simple. In particular, those phases performed in the pipeline require only one subarray computation (one stage) per step, as summarized in Table 4.1. The advantage of this simplicity is that we may examine the

---

Given: a 3×3 subarray in the routing grid

Task: mark traced cells as *blocked* for final-cleanup, *source* for intermediate-cleanup.

Algorithm: Intermediate_Cleanup

```
begin
        if cell in { ↑, ↓, ←, → }
        then cell := free
        else if cell in { trace↑, trace↓, trace←, trace→ }
        then cell := source
end
```

Algorithm: Final_Cleanup

```
begin
        if cell in { ↑, ↓, ←, → }
        then cell := free
        else if cell in { trace↑, trace↓, trace←, trace→ }
        then cell := blocked
end
```

Figure 4.7 One-Layer Local Cleanup Algorithms

---

effects of multi-stage pipelines because we do not need the entire pipeline to perform one wavefront expand or cleanup step.

Phase	Implementation	Function
Wavefront-Expand	1 stage/expand	Expand cells
Backtrace	Host	Trace paths
Intermediate-Cleanup	1 stage/cleanup	Remove expanded cells, mark path as source
Final-Cleanup	1 stage/cleanup	Remove expanded cells, mark path as obstacle

Table 4.1 Local Design for One-Layer RPS Router

4.4.2. Global Design

Global design is concerned with the movement of grids through the pipeline, pipeline programming, and host processing. We treat first the $O(L^2)$ performance problem raised in the analysis of elementary RPS routers, and then treat the system-level data flow in our implementation.

The analysis of wavefront expansion in an RPS pipeline suggests that the method to find a path of length L is to process the routing grid with roughly L/S passes through an S-stage pipeline. This method has three serious flaws:

- **Unreachable, Unexpandable Cells**

  Consider again the example of Fig. 4.4 in which all cells within Manhattan radius L of a source cell are expanded to find a path of length L. Note that the diamond-shaped area of expanded cells occupies at most only 1/2 the entire routing area, that is, only half of all cells can possibly be expanded, but *all* cells must stream through the pipe.

- **Short Wires in Large Grids**

  Most nets in a practical problem occupy much less area than the routing grid representing the entire problem. For example, in gate array routing, 70% to 80% of all nets are shorter than 10% of the chip edge length. Hence, passing the entire routing grid through the pipeline to find short nets is extraordinarily inefficient because of unexpandable cells.

- **Long Wires**

  Some nets in practical problems are necessarily very long, and occupy large regions of the routing grid. Such nets can easily exceed the length of a practical pipeline (10-100 stages) by an order of magnitude. Again, we find the problem of unreachable cells: despite the fact that the final path occupies a relatively large region of the grid, each pipeline pass only extends a wavefront outward a distance $O(S)$ for an S-stage pipe. $L/S \gg 1$ passes are required, and on many passes most grid cells are unreachable and inactive. Thus, intermediate stages of the expansion resemble the problem of short wires in large grids. Moreover, we necessarily revisit labeled (expanded) cells on subsequent pipeline passes. This is of some use with weighted path-functions which occasionally revisit cells, but for our unit-cost router this is only unavoidable overhead.

We attack these three problems by restricting the area of wavefront expansion around a net; this is the notion of *framing*. A variety of frame-like techniques have been applied to software maze-routers, e.g., [Aker72, TYKS80, SNBS84]. These techniques rely on the fact that in practice most nets are routable near their minimum lengths, and do not extend much beyond the minimum bounding rectangle around their points.

Fig. 4.8 illustrates the basic idea of static framing. Around a net's minimum bounding

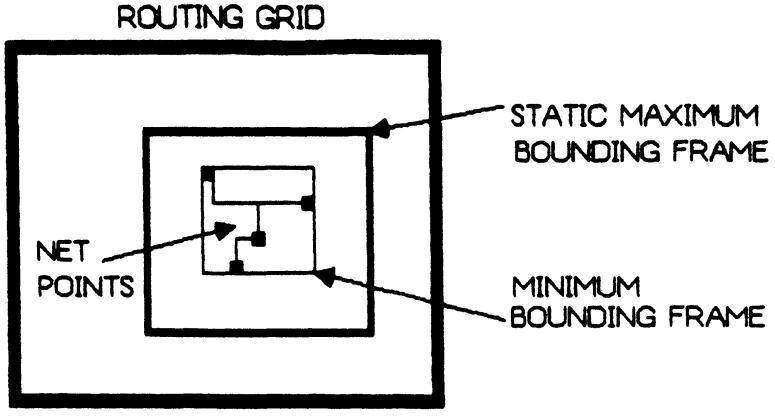


Figure 4.8 Static Framing

frame we size a maximum bounding frame which delimits the area in which cells are to be expanded. These frames are always rectangular. Given a set of net points, we abbreviate their minimum bounding frame as *mBF*, and maximum bounding frame as *MBF*. Static framing mitigates the first problem discussed, that of expanding isotropically from a source: it is rarely necessary to search equally in all directions because, given a source cell, we *know* where the target cells are, and can confine the search to an appropriate frame. This also solves the second problem of short wires in a large grid: the net is framed, and only the frame, not the entire routing grid, is passed through the pipe. This assumes that the pipeline hardware can manipulate arbitrary rectangular subarrays in a cell buffer, which is true for our prototype system.

The third problem, nets long relative to the pipeline length, is addressed similarly with the idea of *incremental framing*. The entire net, large now relative to the routing grid, is again framed as in Fig. 4.8. Consider a 2-point net. Construct a sequence of nested incremental frames, extending outward from the source cell, but never exceeding the bound of the MBF. Wavefront expansions are performed incrementally in each frame as shown in Fig. 4.9. The idea is to dynamically resize a frame just ahead of the expanding wavefront and so minimize the number of unreachable cells processed through the pipe. Frame size determines the time to process each incremental expansion in the pipeline.

There are several important points to note about framing:

- To begin, we need an estimate of the spatial extent of the net to size a static MBF. To determine an incremental framing sequence, we need an estimate of the length of this net which fixes a lower bound on the number of needed wavefront expands.
- Given the unit-cost encoding employed in the local design for the one-layer router, it is necessary that a wavefront never run into a frame boundary: to do so destroys the guarantee that all cells on a wavefront are equidistant from the source. However, wavefronts can run into the MBF since we will never search beyond this boundary.
- If frame $k+1$ is larger than frame k by f cells in each compass direction, no more than f incremental wavefront expands can be performed in frame $k+1$. For an S-stage pipe, $f > S$ implies several pipeline passes per incremental expansion step; $f < S$ implies one

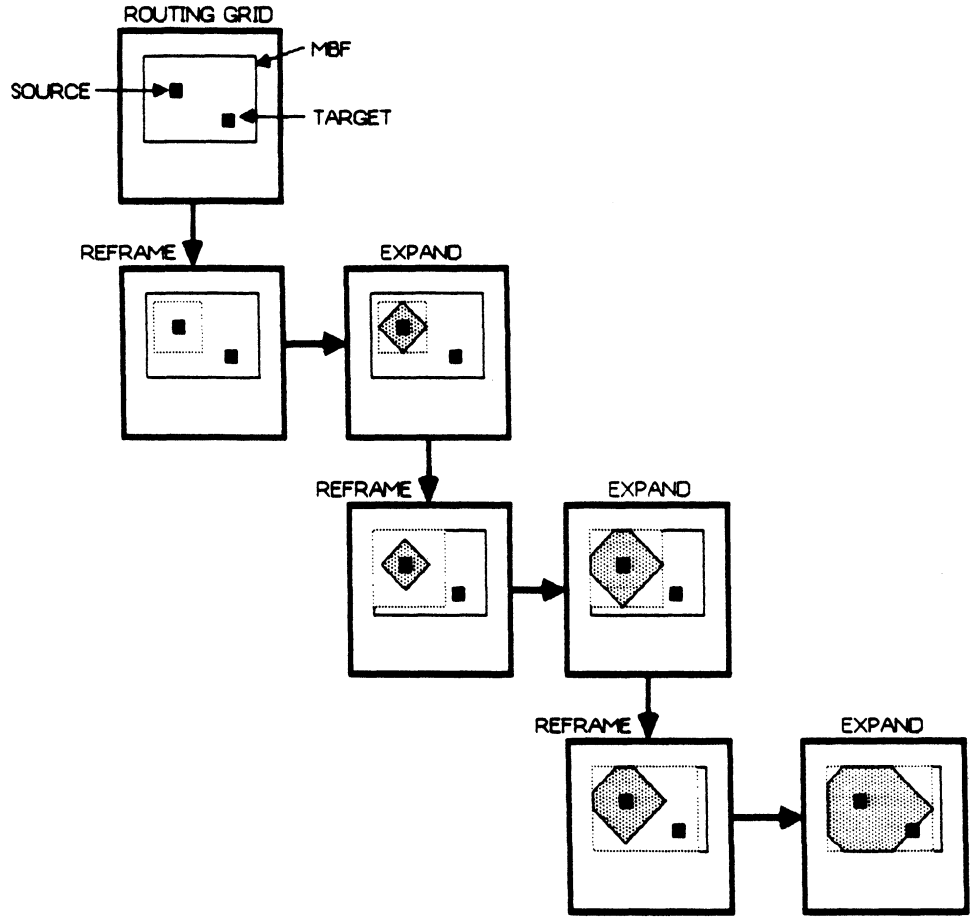


Figure 4.9 Incremental Framing

pass through a shortened pipe.

- Eventually, one frame in an incremental sequence becomes large enough to become the fixed MBF surrounding the net. At this point, there are no restrictions on the allowable number of expansions in this frame since we will never look for a path that crosses the MBF.
- Multi-point nets can be accommodated in an incremental framing scheme also. When a 2-point segment is routed, the first frame is ostensibly just a single *source* cell. Subsequent frames surround it as the expansion wavefront grows outward from the source. For multi-point nets, we start instead with the mBF of those segments already traced, which is

effectively a source *frame* instead of a source cell. Subsequent frames increase outward from this source frame.

- There is host overhead incurred to resize a frame in the RPS hardware. The host must compute the necessary frame size and set it up in the hardware. The time penalty due to communication overhead is large relative to one subarray-computation time. This immediately suggests an optimization problem: how to choose a framing sequence as a function of the spatial extent of a net, the pipeline length and the host overhead.

Incremental framing implies a tradeoff between overhead and pipeline time. A sequence with too many incremental frames, each close in size to the next, minimizes the pipeline time by reducing the number of inactive cells processed, but maximizes the overhead of reframing. Choosing instead fewer frames minimizes framing overhead, but maximizes the pipeline time by processing wasted cells.

Given a multi-point net, the choice of MBF, sequence of incremental frames, and number of trial expansion steps constitutes a *framing strategy.* Our implementation parameterizes a strategy as follows.

- If the minimum bounding frame for the net has dimensions X_{min}, Y_{min}, a static maximum bounding frame is constructed by extending the minimum frame a distance X_{ext} in both horizontal directions, Y_{ext} vertically, where:

$$X_{ext} = a_x X_{min} + b_x \ , \ Y_{ext} = a_y Y_{min} + b_y \tag{4.4}$$

  Let X_{left}, X_{right}, Y_{top}, Y_{bottom} define a frame. Then:

$$\begin{aligned} X_{left}^{max} &= X_{left}^{min} - X_{ext} \ , \ X_{right}^{max} = X_{right}^{min} + X_{ext} \\ Y_{top}^{max} &= Y_{top}^{min} - Y_{ext} \ , \ Y_{bottom}^{max} = Y_{bottom}^{min} + Y_{ext} \end{aligned} \tag{4.5}$$

- Incremental frame $k+1$ is constructed from frame k by extending it at most f cells in each compass direction, subject to the constraint that it never extends past the MBF. Incremental frames are bounded by the maximum frame. f is the *frame-increment,* the maximum allowable incremental expansion allowed in one frame which determines how many pipeline passes are required to process expands in each frame. This defines a very simple heuristic: *constant-increment, isotropic framing.* Frame $k+1$ has dimensions:

$$X_{left}^{k+1} = \min(X_{left}^{k} - f,\ X_{left}^{max}),\quad X_{right}^{k+1} = \min(X_{right}^{k} + f,\ X_{right}^{max})$$
$$Y_{top}^{k+1} = \min(Y_{top}^{k} - f,\ Y_{top}^{max}),\quad Y_{bottom}^{k+1} = \min(Y_{bottom}^{k} + f,\ Y_{bottom}^{max}) \tag{4.6}$$

Frame $k+1$ is the last frame in the sequence when it has the same dimensions as the MBF.

- The number of trial wavefront expands is just the estimated net length for a 2-point net. For a multi-point net, we can estimate the length of individual segments, or estimate the length of the entire net. Expected net length L_{exp} is computed as

$$L_{exp} = a_{len}\, L_{mst} \tag{4.7}$$

where L_{mst} is the length of the minimal Manhattan spanning tree for the net points, and a_{len} is an empirical constant that estimates how far a net is likely to meander past its minimum length. Note that it is important to estimate net length accurately because the RPS hardware cannot determine when a target has been expanded; the host must go check the grid. It may be the case that a target is expanded in the middle of a pipeline, and we cannot simply stop pipeline data flow in mid-raster. This derives from the fact that an RPS pipe performs several local-instructions (wavefront expands) in parallel as a unit. In such a case, we shall overrun the target and perform too many expands. The local design for the one-layer router is such that this will not disturb the optimal path found. Hence, a good net length estimate is essential.

For specific RPS hardware, $\{a_x, b_x, a_y, b_y, a_{len}, f\}$ constitute a set of tunable parameters for optimizing system performance. As a common example, we might size the MBF to be 15% larger than the mBF of the net, estimate net length to be at most 10% longer than its minimum Manhattan length, and determine the frame-increment f experimentally.

Algorithm Route_1_Net (Fig. 4.10) summarizes the global algorithm for routing multi-point nets. Steps requiring the RPS hardware are marked **<operation>** to distinguish them from steps performed solely on the host. Figure 4.11 outlines the host/hardware data flow graphically. Note that the expansion phase terminates when either all targets are expanded, for which the host must explicitly check, or after exceeding some fixed threshold of expand steps, after which some targets are yet unexpanded. Failure to find a path can be dealt with in several

Given: a multi-point net, and a routing-grid.

Task: route the net in the grid in incremental frames on RPS hardware.

Algorithm: Route_1_Net

```
begin
        estimate L_exp expected net length
        l := L_exp
        <mark one net point as source, all others target>
        source_frame := single source cell

        for p := 1 to number_of_targets do
        begin
                compute framing strategy based upon
                   source_frame and remaining net length l
                found := false
                while ( more frames ) and ( not found ) do
                begin
                        <resize frame in RPS hardware>
                        <expand frame in pipeline>
                        if ( active targets in frame )
                        then begin
                                <return active target cells>
                                if ( any target expanded )
                                then found := true
                        end
                end

                if ( not found ) then
                begin
                        <do final_cleanup in pipeline>
                        STOP
                end

                <return current frame to host>
                backtrace shortest path
                source_frame := mBF( current traced net segments )
                l := l - new_path_length
                <return frame to RPS hardware>
                if ( last target )
                 then   <do final_cleanup in pipeline>
                 else   <do final_cleanup in pipeline>
        end
end
```

Figure 4.10 One-Layer RPS Router Global Algorithm

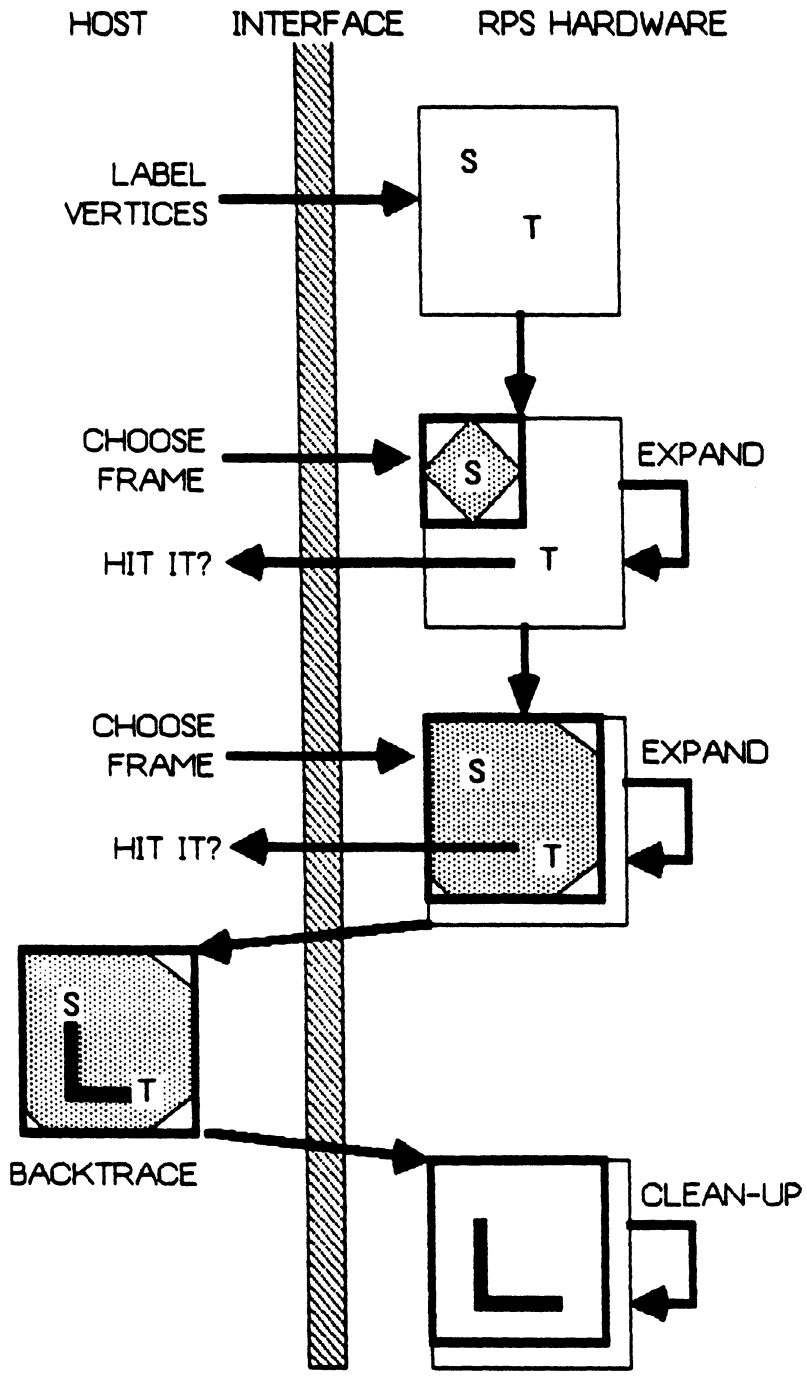


Figure 4.11 Global Data Flow for One-Layer Router

ways: choose a new larger maximum frame; remove the constraining maximum frame entirely; increase the net length estimate; or quit. The current implementation permits the user to choose any of these options.

A multi-point single-layer router embodying all these ideas has been implemented in the prototype RPS environment. The following sections examine its performance experimentally.

4.4.3. Framing Experiment

We first measure the effects of the constant-increment framing heuristic. A standard routing problem has been defined: a 2-point orthogonal L-shaped net of length L=256 cells centered in an empty 512 × 512 grid. This is one net in the set of nets chosen for performance benchmarks in the following section; see Fig. 4.13. The maximum bounding frame is chosen approximately 25% larger than the minimum bounding frame. As an experiment, we vary the frame increment f and the pipeline length S. Note there are roughly L/f frames in a strategy: small f yields many frames with few wavefront expands done in each; large f yields few frames with many expands each. Fig. 4.12 plots elapsed routing time as a function of frame increment

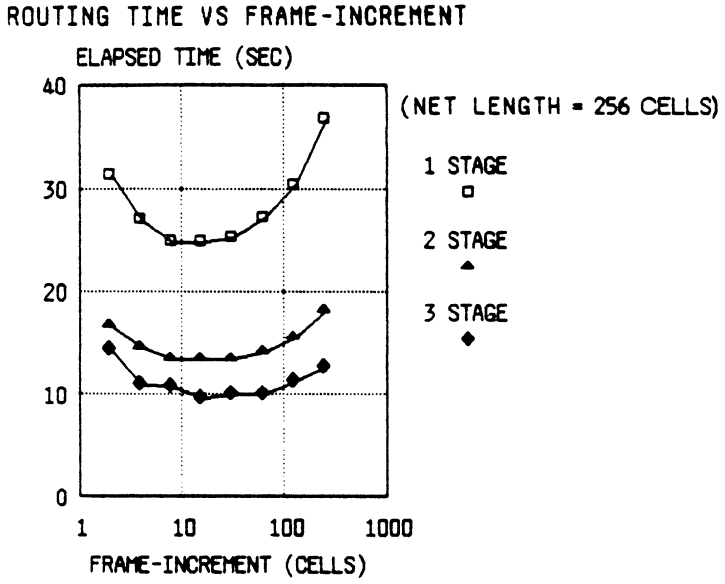


Figure 4.12 Elapsed Routing Time vs. Frame-Increment, Pipeline Length

f and pipeline length. For each pipeline length, the elapsed-time curve has a global minimum. Too many frames is costly because of overhead; too few frames costly because of processing unexpandable cells. This cost is as high as $\approx 50\%$ in this experiment, which is unacceptable on a per wire basis for a router. This experiment establishes that sub-optimal framing is detrimental to execution time. These results are used to empirically optimize the performance of the router.

4.4.4. Single-Net Experiments

Having established that the effects of framing are significant, we now focus on the effects of pipeline length and net length. A family of standard routing problems is studied: 2-point nets of length 2^k cells, $k = 2, 3, ..., 9$, and 4-point nets of length 2^k cells, $k = 4, 5, ..., 9$, centered in an empty 512×512 grid. Elapsed and host-CPU times for the router are measured for each problem. Although embedding one net in the center of an empty grid seems trivial, recall that a maze-router embeds one net at a time, and that grid congestion does not affect the routing time *for a constant net geometry.* More precisely, the time to route a net of a given shape and length within a given MBF is *constant*, independent of its surroundings. Congestion will of course change a routing path, or block it, but for a constant shape--such as those in our experiment--time is fixed. Hence, this experiment is a good benchmark by which to predict performance. Fig. 4.13 illustrates the test nets, and plots routing times for these nets, averaged over several runs.

As expected, elapsed time increases with net length, and decreases with pipeline length. The data for extremely short nets is noisy because the elapsed time is dominated by host overhead. For longer nets, elapsed time decreases roughly linearly with pipeline length, i.e., an S-stage pipe is S times faster than a 1-stage pipe. Host-CPU time measures primarily the overhead incurred by pipeline commands and grid IO, notably the time to get the current frame for host backtrace. It is fairly flat, and increases slightly for longer nets (because more pipeline passes/net implies more pipeline commands) and decreases slightly with longer pipelines (because fewer passes/net implies fewer pipeline commands).

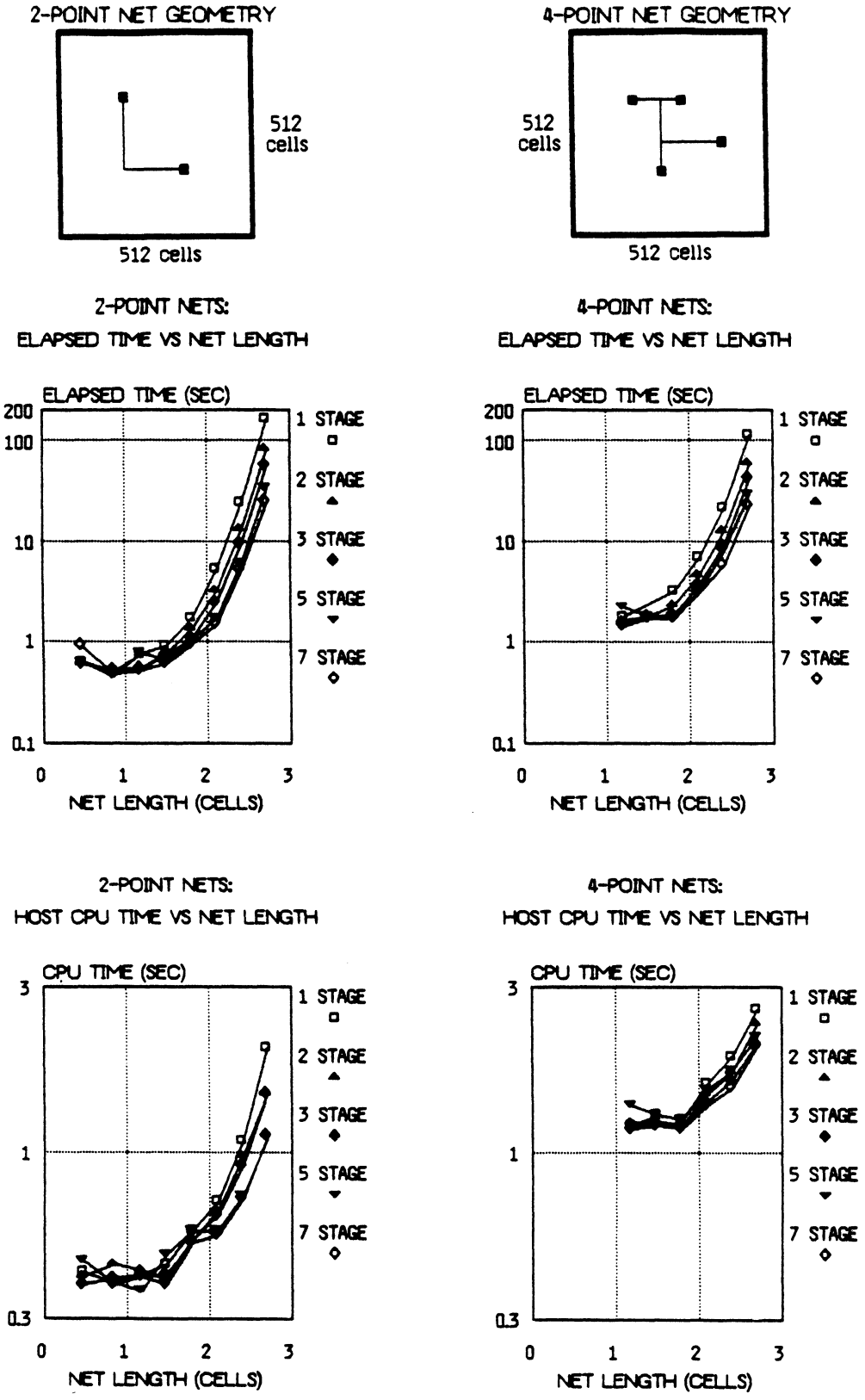


Figure 4.13 One-Layer Single-Net Routing Benchmarks

Note that these curves can be used to extrapolate how many stages are needed to route a net with length less than L_{max} in time less than T_{max}. For example, given $L_{max}=256$, 3 stages suffice to meet $T_{max}=10$ seconds, approximately 30 stages to meet $T_{max}=1$ second.

4.4.5. PCB Experiments

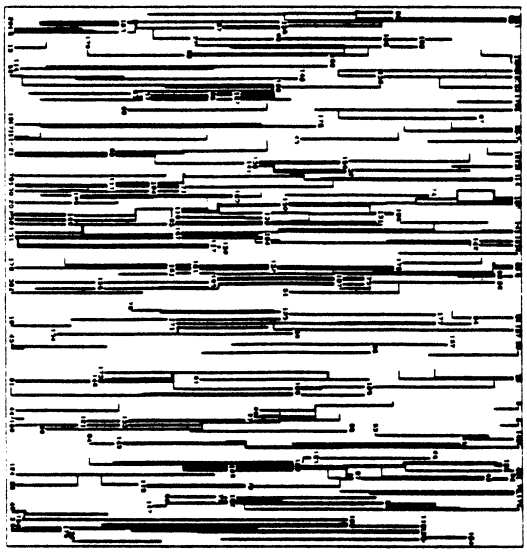
Consider now the more practical problem of Printed Circuit Board (PCB) routing. We compare the performance of the single-layer RPS router with a conventional software maze-router running on several machines. We use the PCB benchmark and software maze-router developed by Blank in [Blan82]. The task is to embed 200 predominately right-way nets (intended to represent a single layer of a board) with average length $\approx$ 190 cells in a 512×512 grid.

The software maze-router is written in Pascal and embodies some of the ideas of [Hoel76]. This experiment compares the RPS router running on 3-stage and 6-stage pipelines against this software running on several machines; see Table 4.2. For fairness, each machine is lightly loaded.

Machine	MIPS (approx)	Op Sys	Load (approx)
Amdahl 5860	12	MTS	$\sim$ 100 users
DEC VAX 11/780	1	UNIX	load fac. $\sim$ 1-2
DEC VAX 11/750	0.6	UNIX	load fac. $\sim$ 1
Apollo DN600	0.5	AEGIS	single-user
VAX 11/780 + 3-stage RPS pipe	-	UNIX	load fac. $\sim$ 1-3
VAX 11/780 + 6-stage RPS pipe	-	UNIX	load fac. $\sim$ 1-2

Table 4.2 Machines for RPS/Software Maze-Router Comparison

The routed PCB result is shown in Fig. 4.14, along with a plot of elapsed time and CPU time for the software and RPS routers. CPU time for the RPS router is host CPU time; elapsed time is wall-clock time. Elapsed time is included in the basic comparison because it is the most appropriate metric for dedicated hardware. Elapsed time for the RPS router is superior to the elapsed and CPU times of all except the largest mainframe. We estimate that between 9 and 12



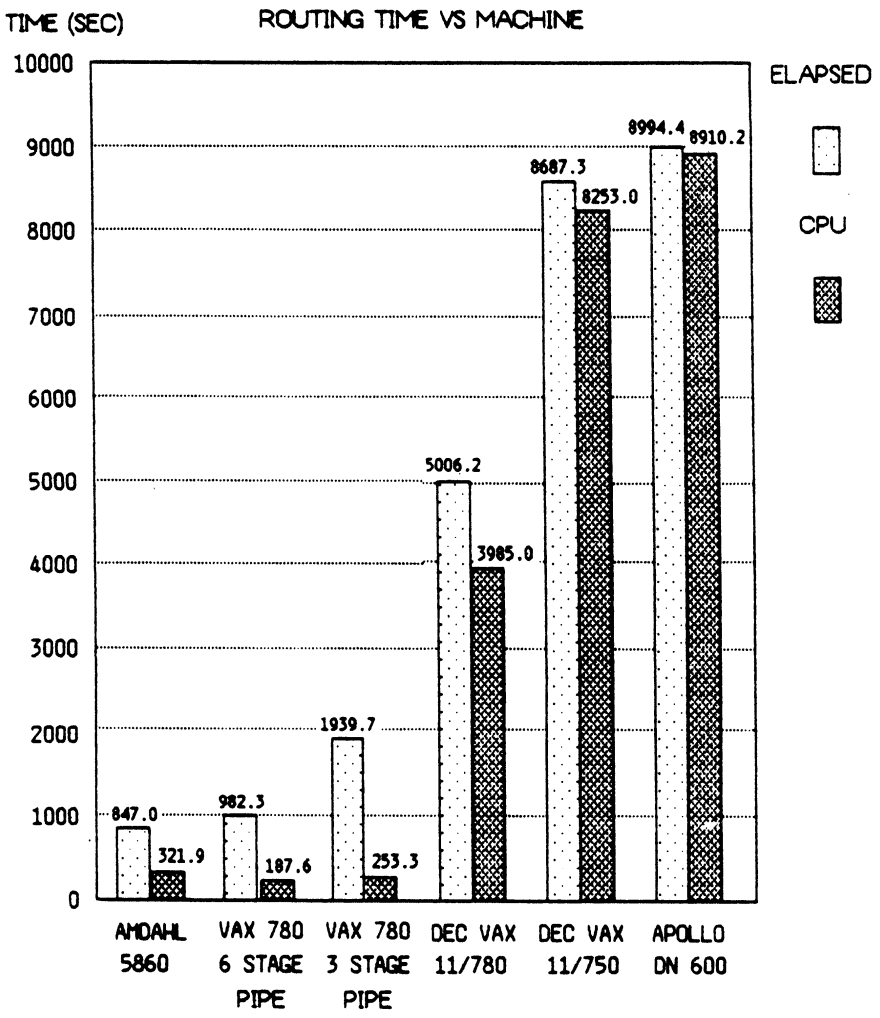


Figure 4.14 RPS and Software Routers Compared for PCB Benchmark

RPS stages would surpass the mainframe performance. Moreover, the RPS router requires less CPU time than any software implementation. We take this to imply that it is not burdensome to the host to manage the RPS hardware.

CPU time subtracted from elapsed time is a rough estimate of pipeline processing time[3]. Ideally, this quantity decreases as $1/S$ for an S-stage pipe. With host overhead, this quantity decreases less rapidly. In this experiment, the ratio of rough pipeline time between the 6-stage and 3-stage systems is 2.14, reflecting a slightly higher host loading during the 3-stage run.

4.5. Two-Layer Routing

This section constructs an RPS router to embed 2-point nets in two interconnect layers with floating vias. The one-layer router previously discussed has a simple local design--one stage for expand, cleanup--in order to study the effects of varying pipeline length over a modest range (1-7 stages). This simplicity weakens its effectiveness as a practical router. To remedy this, we consider the local and global design of a more complex router, and test its effectiveness on real PCB and gate-array tasks.

4.5.1. Local Design

The cell encoding for a two-layer router must support wavefront expansion on separate layers, and vias between layers. We allow vias to *float*, to appear wherever they are not prohibited in the grid. Via exclusion is the process of prohibiting illegal via adjacencies in the grid; exclusion prevents new nets from embedding vias too close to those in previously routed nets. Two-layer routing is conventionally divided into separate horizontal and vertical layers, where each layer is said to have a *preferred* direction.

Wavefront expansion is more complex with two layers. A *source* now expands in separate wavefronts on each layer. However, this expansion cannot now proceed isotropically, as in Fig. 4.4, because net segments can no longer follow an arbitrary direction. One approach is to permit only horizontal or vertical segments on a given layer. This has the disadvantage of

[3]A rough estimate because of the time-shared host. Elapsed time includes host processing for the RPS router, pipeline time, IO, and some dead time in which no portions of the RPS router are running.

introducing many vias in most nets. A compromise is to permit segments to jog short distances in the non-preferred direction to reduce the need for vias.

Again, we design a unit-cost router using an alphabet of source-pointing arrows. The one-layer encoding is extended so each cell is now a 3-tuple containing via, layer-1, and layer-2 information. Layer-1 is mostly horizontal, layer-2 mostly vertical. The alphabet Λ_{2layer} is the Cartesian product of these three classes of labels $\Lambda_{horiz} \times \Lambda_{vert} \times \Lambda_{via}$:

$$\begin{aligned}
\text{Layer-1 Horizontal:} \quad & \Lambda_{horiz} = \{ \leftarrow h0, \rightarrow h0, \leftarrow h1, \rightarrow h1, \uparrow h1, \downarrow h1, \uparrow h2, \downarrow h2, \\
& \qquad h\text{-}via,\ h\text{-}free,\ h\text{-}blocked,\ h\text{-}src \} \\
\text{Layer-2 Vertical:} \quad & \Lambda_{vert} = \{ \uparrow v, \downarrow v, \leftarrow v, \rightarrow v,\ v\text{-}via,\ v\text{-}free,\ v\text{-}blocked,\ v\text{-}src \} \\
\text{Vias:} \quad & \Lambda_{via} = \{ via,\ novia \}
\end{aligned}$$

Similar to Λ_{1layer}, each layer has source, target, and blockage symbols, and expansion again proceeds with wavefronts of source-pointing arrows. This encoding restricts jogs as follows: one net segment on one layer can jog exactly once a short distance in the non-preferred direction. Horizontal segments can jog 2 cells, vertical segments 1 cell. Expansion with preferred directions and these jog restrictions is contrasted with isotropic expansion in Fig. 4.15. The figure shows the 24 ways in which a segment can escape from a source on the two layers, illustrating the possible jogs; segments are labeled with the appropriate symbols from this alphabet. Vias appear as labels on each layer. Vias can be generated when a cell is expanded on one layer and free on the other. If a via is not prohibited with the *novia* symbol, the other layer is relabeled as a new source, and expansion proceeds there. These particular path restrictions permit a compact pipeline implementation. Unlike the one-layer router, the two-layer router requires more than a single subarray-computation.

The local design resembles that of the one-layer version except for its greater complexity. Wavefront expansion now requires three subarray-computations, thus diminishing the number of expand steps performed in an S-stage pipeline to $S/3$. Backtrace is again done on the host, and is similar to the one-layer version except that a trace can cross layers at a via. Wire segments are relabeled as *blocked* on the appropriate layer, and vias are labeled with the 3-tuple (*via, h-via, v-via*). Cleanup performs the standard removal of extraneous arrow labels, and also enforces via exclusion. Cells adjacent in the four compass directions to a newly traced via are

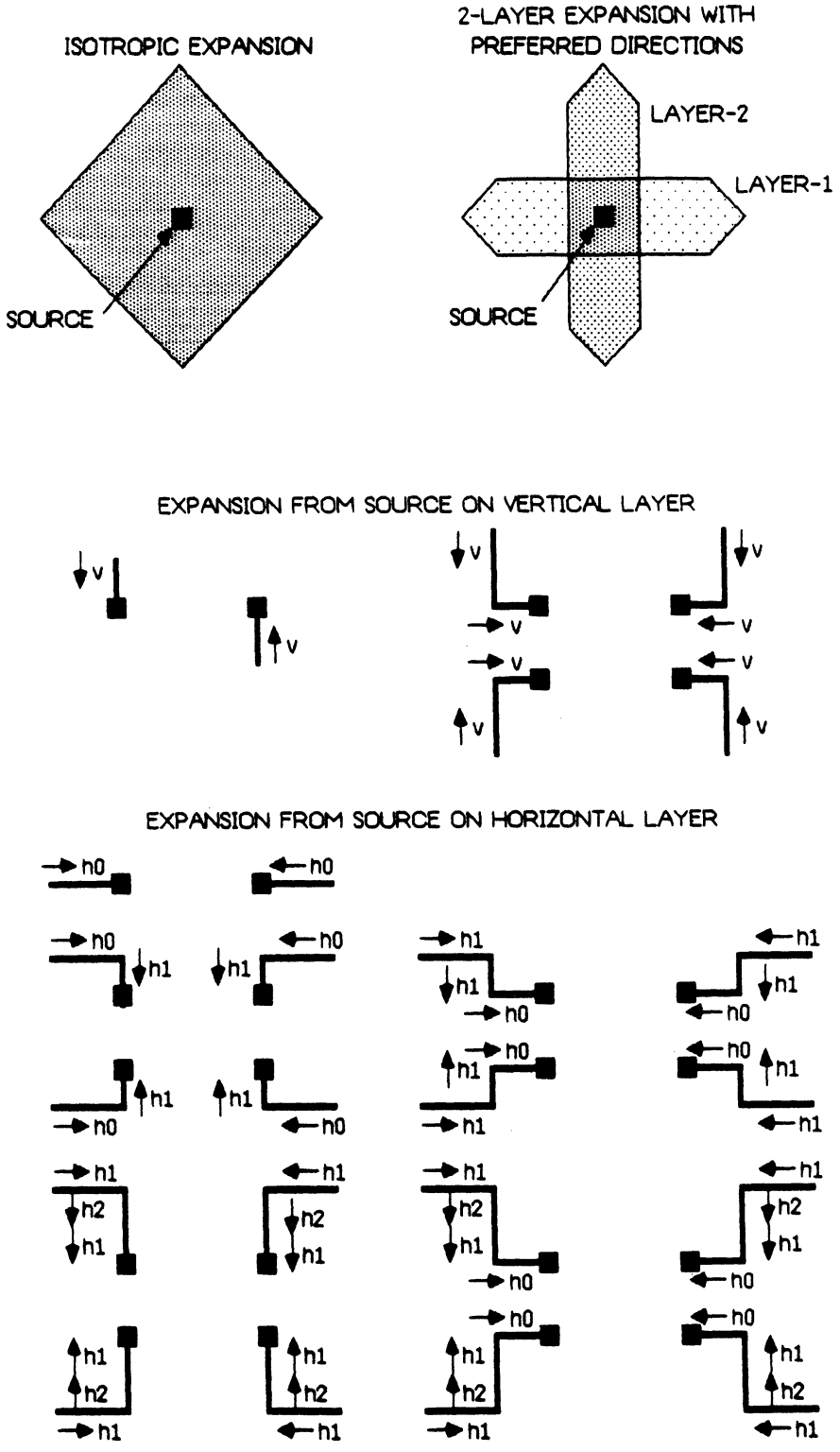


Figure 4.15 Two-Layer Expansion with Preferred Directions, Jogs

marked *novia* so that future vias are not embedded too close. This is a common physical restriction for vias in PCB or VLSI technologies. Cleanup requires two subarray-computations. Appendix A presents these local algorithms in detail. Table 4.3 summarizes the local design.

Phase	Implementation	Function
Wavefront-Expand	3 stages/expand	Expand cells in 2 layers with preferred directions, vias
Backtrace	Host	Trace paths
Cleanup	2 stages/cleanup	Remove expanded cells, mark path as obstacle on appropriate layers, perform via-exclusion

Table 4.3 Local Design for Two-Layer RPS Router

4.5.2. Global Design

Global design for the two-layer router is essentially identical to that of the one-layer router. The advantage of the local/global partition is the ability to re-use major system components while changing details of the pipeline processing. In this case, the same global algorithm, incremental framing and data flow apply. The code running on the host is a modification of the one-layer code to manage the more complex pipeline programming. Only 2-point nets are supported in this version.

Similar framing and single net experiments are performed to verify and tune system performance; these are not discussed.

4.5.3. PCB and Gate Array Experiments

We now examine how an RPS router might be applied to realistic problems by integrating it into a standard place-and-route environment. Figure 4.16 shows the major components of a simple environment developed to support these experiments. This system places a set of modules and signals using a standard pairwise interchange technique [HaKu72], binds the abstract placement to the geometry of a physical carrier, routes the carrier using the two-layer RPS router, and displays and analyzes the results. Binding to geometry produces the routing-

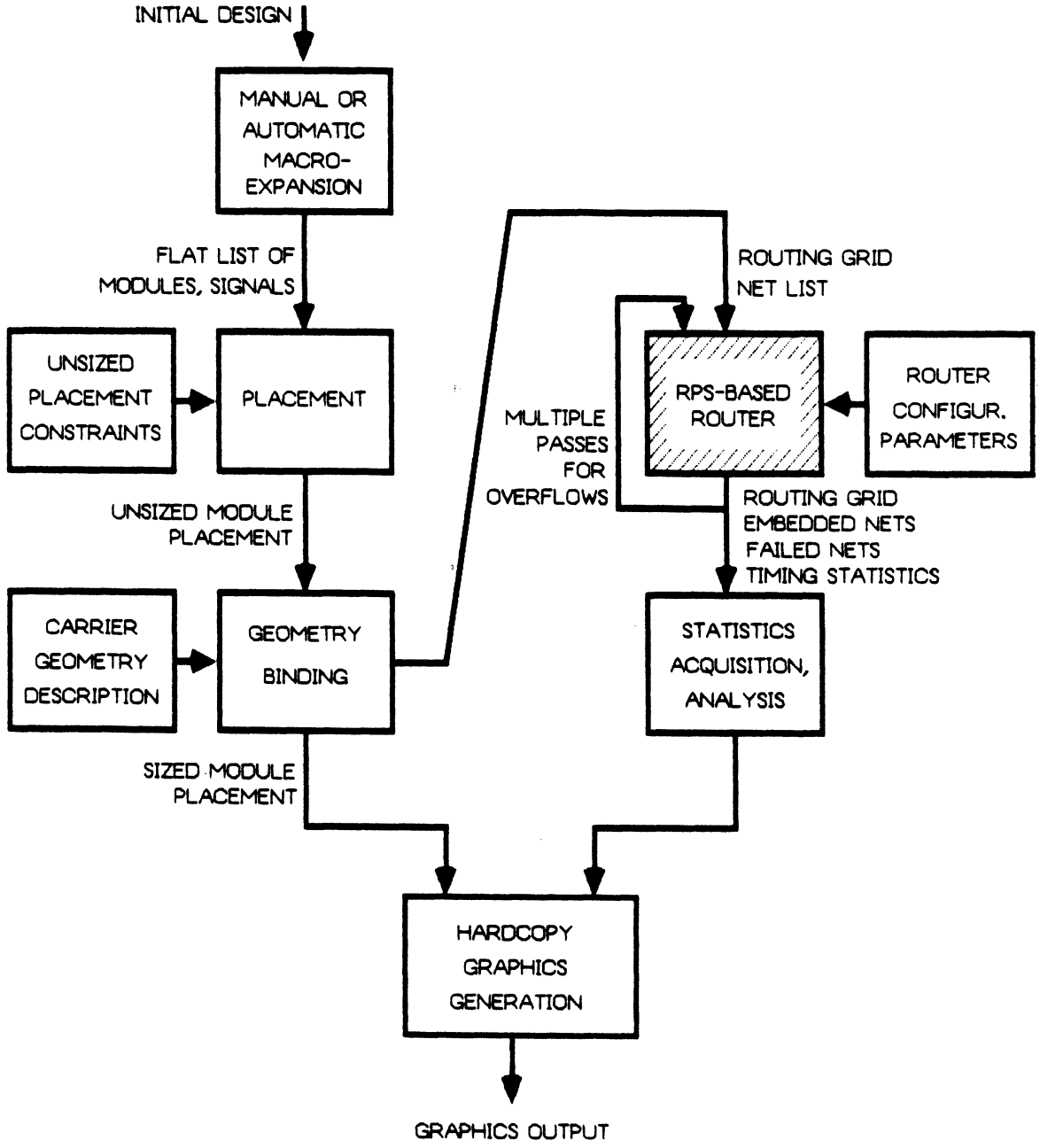


Figure 4.16 Place and Route System Environment

grid and is basically a task-transformation as discussed in Chapter II. We study only the performance of the RPS router in this system. Two experiments are performed using a PCB and a gate array.

The PCB experiment places 29 TTL packages of varying shapes on an 10 × 12 inch board. The PCB is routed in two phases. First, via-availability is reduced near pins to prevent pin-blocking of nets. The grid is routed and unroutable overflow nets are recorded. The grid is then processed to restore via-availability near module pins. This is again a simple task-transformation. The second phase then embeds these overflow nets. The routed result and performance for each phase appear in Figure 4.17.

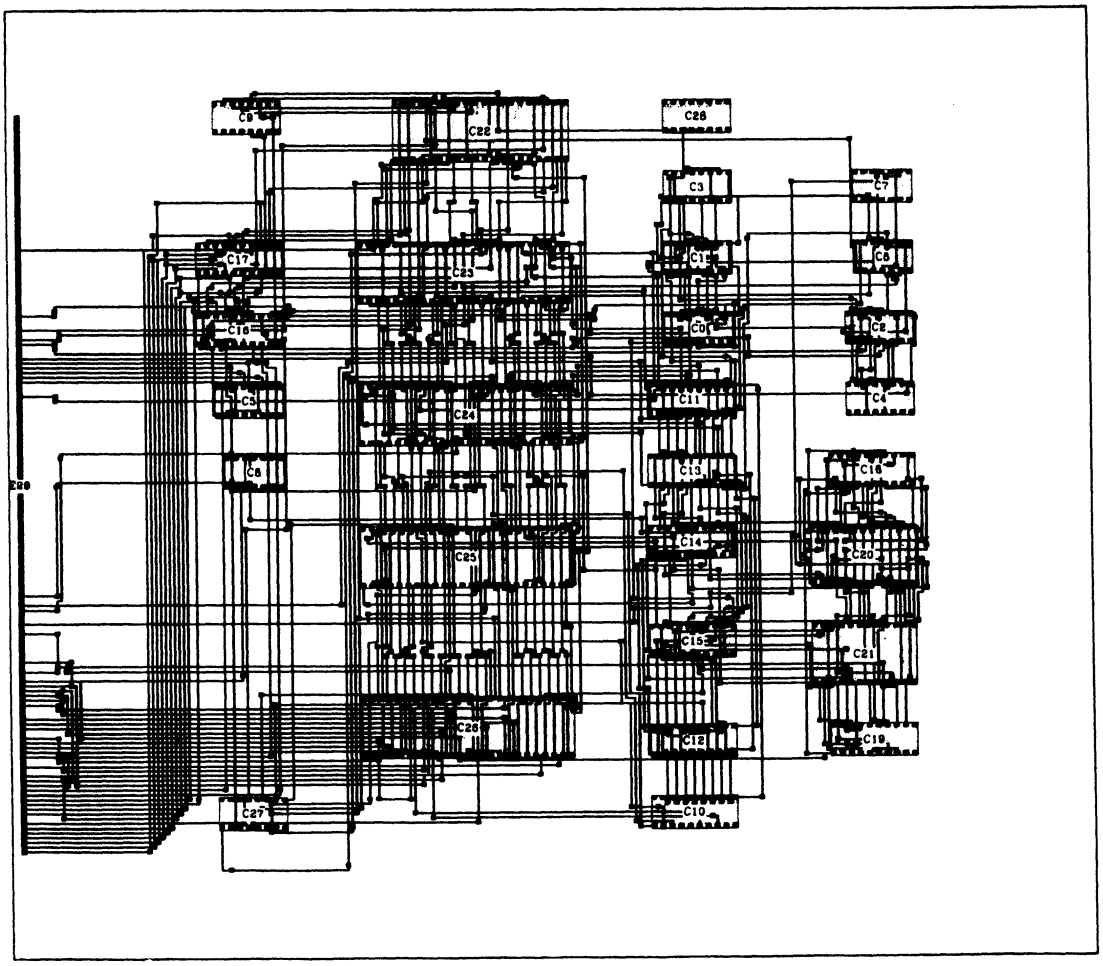
The gate array experiment places a 735-gate circuit in a 900-gate array. The circuit represents a simplified portion of an 8-bit ALU with registers; the 82% gate occupancy of the array is fairly typical. We have designed a gate array image following closely realistic chips. The gate array is represented as a 25 × 18 array of circuit blocks, where each block holds up to two 4-input gates. Vertical channels provide 12 wiring tracks, horizontal channels 7 tracks, and unused blocks afford 3 extra vertical tracks. We do not route IO pads, but do fix some driver gates around the periphery of the array. Routing is done in one pass, and overflows are ignored here. Figure 4.18 gives the performance for 3-stage and 6-stage pipelines and shows the routed result.

As a further experiment, we have instrumented the 3-stage two-layer router to give a more accurate breakdown of the elapsed time. We measure $T_{GA}(S, \tau_{stage})$, the time to route the gate array on an S-stage pipeline with stage cycle time τ_{stage}, and decompose it into the following times:

$$T_{GA}(S, \tau_{stage}) = \tau_{host} + \tau_{backtrace} + \tau_{cleanup} + \tau_{expand} \qquad (4.8)$$

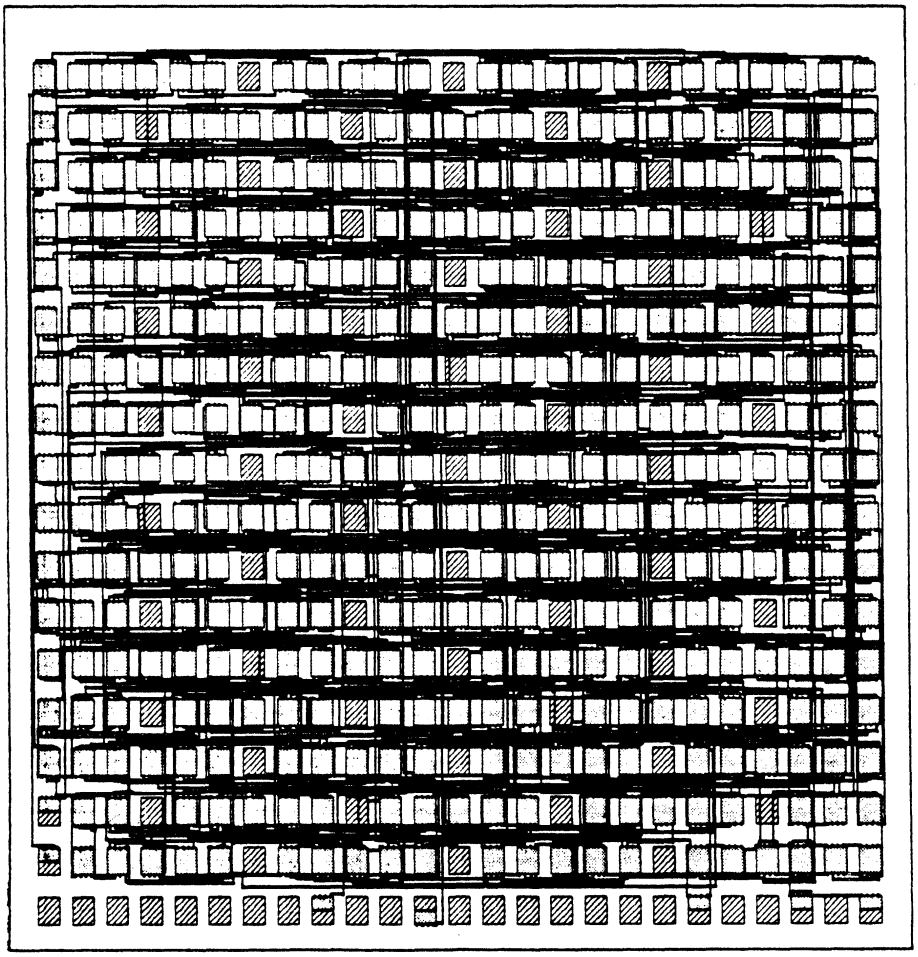
We have measured $T_{GA}(3, 2\mu s)$ experimentally and measured or estimated these component times. To estimate $T_{GA}(S, \tau_{stage})$ in general, we make these assumptions:

- τ_{host} is constant for a given host and includes actual CPU processing to support strategy calculation and expansion, some IO time, instrumentation overhead, and general time-sharing overhead. (Actually, τ_{host} varies with host loading, and also slightly with pipeline



PCB Task Parameter	Two-Layer 3-Stage RPS Router		
	Phase 1	Phase 2	Total
Grid Size	200 X 240 cells	same	same
Nets Tried	412	81	412
Nets Completed	331	41	372
%Completions	80%	51%	90%
Length Routed (50 mil/cell)	19323 cells 966.2 in.	3746 cells 187.3 in.	23069 cells 1153.5 in.
Elapsed Time	22.95 min.	11.05 min.	34.0 min.
CPU Time	4.24 min.	1.35 min.	5.59 min.

Figure 4.17 Two-Layer PCB Experiment



Gate-Array Task Parameter	RPS Two-Layer Router	
	3-Stage	6-Stage
Grid Size	500 × 502 cells	same
Nets Tried	1155	1155
Nets Completed	1087	1086
%Completions	94%	94%
Length Routed ($8\mu m$/cell)	69966 cells 0.56 m	69907 cells 0.56 m
Elapsed Time	71.58 min.	50.99 min.
CPU Time	10.43 min.	11.12 min.

Figure 4.18 Two-Layer Gate Array Experiment

length and net length, which we ignore here; see CPU times in Fig. 4.13.)

- $\tau_{backtrace}$ is constant for a given host, and includes actual trace processing and IO to retrieve the frame from the hardware.
- $\tau_{cleanup}$ is proportional to τ_{stage} because cleanup is dominated by pipeline processing time and requires only a few stages.
- τ_{expand} is proportional to τ_{stage}, and inversely proportional to $S/3$ for a 3-stage 2-layer expand step. That is, we assume τ_{expand} is entirely pipeline processing time, and hence that a faster stage reduces time proportionally, as does a longer pipeline.

Given this data for $T_{GA}(3, 2\mu s)$, then $T_{GA}(S, \tau_{stage})$ can be computed as:

$$T_{GA}(S, \tau_{stage}) \approx \tau_{host} + \tau_{backtrace} + \tau_{cleanup}\left\lceil \frac{\tau_{stage}}{2\mu s} \right\rceil + \frac{\tau_{expand}}{S}\left\lceil \frac{\tau_{stage}}{2\mu s} \right\rceil \qquad (4.9)$$

Fig. 4.19 uses this formulation and plots T_{GA} for longer and faster pipelines. This is a best-case estimate and does not account for expected variations in τ_{host}. As expected, longer pipelines and faster stages give better performance. Host overhead, notably the time to retrieve a frame for backtrace, is a bottleneck in this implementation, producing the asymptote shown in the figure. The predicted 6-stage time differs from the measured 6-stage time by about 15% due to variations in τ_{host}.

Some discussion of the performance of the two-layer router, relative to software maze-routers, is appropriate here. The conventional metrics are execution time and net completions. [High83, Wils83] cite times of 30 CPU minutes on a 1 MIP machine for maze-routing 1K-2K gate arrays. The data in Fig. 4.19 indicate that for our not-quite-1K gate array, this time is easily bettered by modestly longer or faster pipelines. The completion rates (90-94%) for the RPS router are low, again due mainly to the local design. The PCB experiment uses a simplistic model of an edge-connector which forces signals to escape on one layer and accounts for most of the failed connections. In both experiments pin-blocking is a problem. Nets escaping from pins jog several times and consume via-sites, thus blocking later nets from escaping. A potential solution is a via-optimization pass during backtrace [High83]. Unneeded vias are removed as short wrong-way segments are relocated to other layers. Nevertheless, to improve

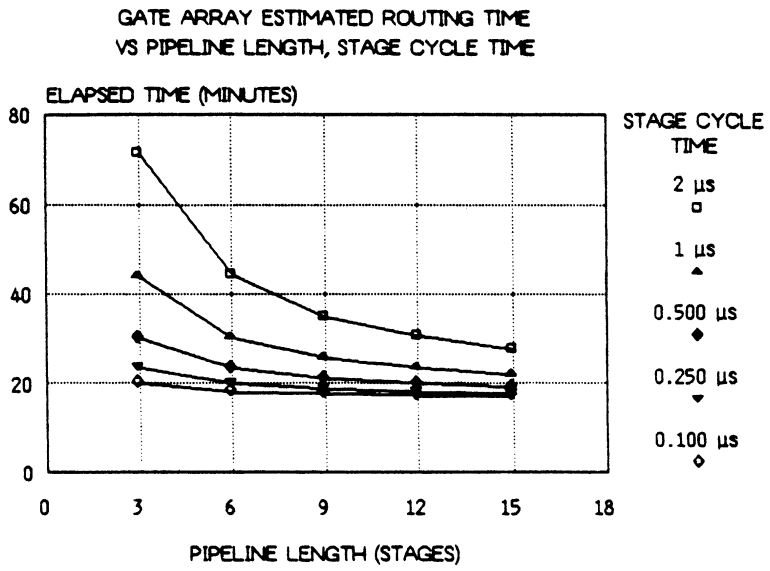


Figure 4.19 Estimated Gate Array Routing Times

routing quality further, the hardware must accommodate more complex local algorithms, e.g., weighted path-functions. To address larger problems directly, e.g., 2K to 10K gate arrays, we need only add memory to the cell buffers; even large grids can be accommodated in the subarray stages. [High83] cites 13.5 Mbytes as the storage necessary to maze-route a 10K gate array, an amount which is not impractical for dedicated hardware given current memory technology. The superior routing quality achievable by more intelligent maze-routers, along with the practical speedups suggested by Fig. 4.19, together constitute a compelling argument for an RPS architecture designed specifically to support routing.

4.6. Global Routing

Global routing assigns nets to routing regions rather than to detailed tracks. The one-layer and two-layer routers are detailed routers by contrast. This section discusses the design of an RPS global router. Global routers are widely used as the first phase for VLSI routing, e.g., gate arrays, followed by detailed routing of channels.

Global routing is also attractive as a first phase for a subsequent RPS detailed router. Long wires are more expensive to route than short wires in an RPS pipe. Even with a good

placement, a few wires span the entire routing grid, and are disproportionately expensive given the $O(L^3/S)$ time complexity of RPS routing. Incremental framing is a first-order solution, but the last pipeline passes that expand a long wire still process very large grids. Global routing offers a solution as shown in Fig. 4.20. Instead of routing these directly, we partition the routing grid, perform first a global routing of nets across routing regions, and then a detailed routing of net segments in each region. This reduces these worst-case longest nets and improves execution time[4]. Moreover, this hierarchical routing scheme offers other practical advantages. First is the prospect of better routing quality because of global topological optimization. Second, partitioning into smaller routing tasks means that smaller RPS cell-buffers will suffice for each, and that there is more potential for parallelism as each task may be routed concurrently if parallel pipelines exist.

4.6.1. Local and Global Design

We treat the design of a simple gate-array-style global router. The routing grid is again a rectilinear grid, but each cell is now a routing region. Cells contain resources consumed by

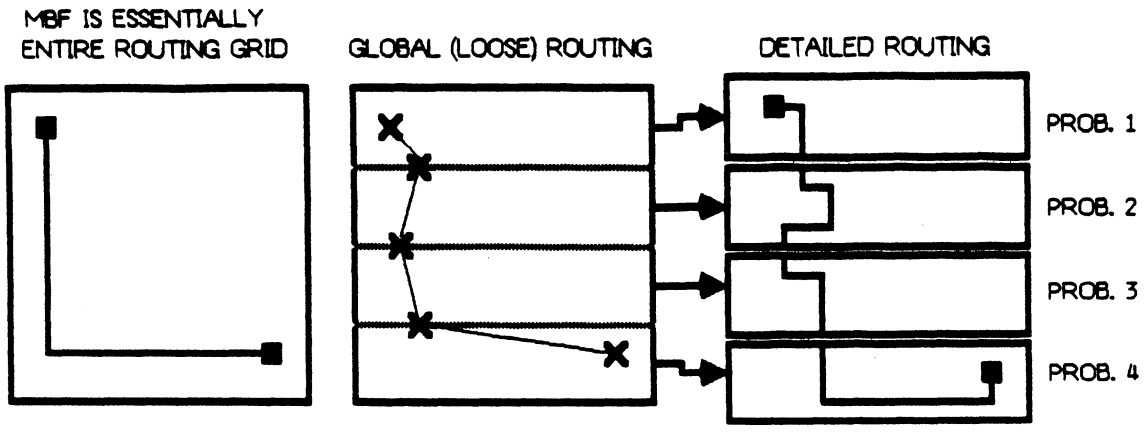


Figure 4.20 RPS Global Routing Followed by RPS Detailed Routing

[4]This is roughly analogous to framing a net to reduce the area searched. However, a smaller routing region does not *guarantee* a shorter net: the best path between two close points may still be long (see [Souk81]). On the average, however, this approach will reduce the problem.

paths that cross them. It is now the edges of the cells which are of interest: these are the *routing channels* crossed by nets. Channels have a fixed capacity of tracks for nets. Ideally, we route with a path-function that penalizes nets that cross congested channels, or enter a cell from one direction and exit from another to consume a via. Cytocomputer stage architecture again prohibits this sophistication. Instead, we design a global router that finds shortest paths for 2-point nets through admissible channels. A channel is admissible if it has a non-zero track capacity.

Local and global design are summarized here. As before, wavefront expansion employs source-pointing arrows in one conceptual layer. Cells are now characterized by their boundary channels which have a capacity of 0 to 15 tracks; each cell stores two channels, its neighbors store the remainder.

Fig. 4.21 illustrates the global data flow. Starting with a grid with channel capacities, there is an initial translation phase: channels with non-zero capacities are marked as *free* channels, others are marked as *blocked* channels. This new grid is the *binary-capacity* grid. Wavefront expansion proceeds as with the one-layer router except that channels, not cells, are crossed. When a source-to-target path is found, the grid is returned to the host for backtrace. Subsequent cleanup produces another grid indicating channels with capacity diminished by the newly traced path. This Δ-*capacity* grid is returned to the host. Cell-by-cell subtraction of the new Δ-capacity grid from the original routing grid updates the routing grid to reflect the new path. Routing the next net then proceeds. Detailed algorithms appear in Appendix B. Table 4.4 summarizes the local design.

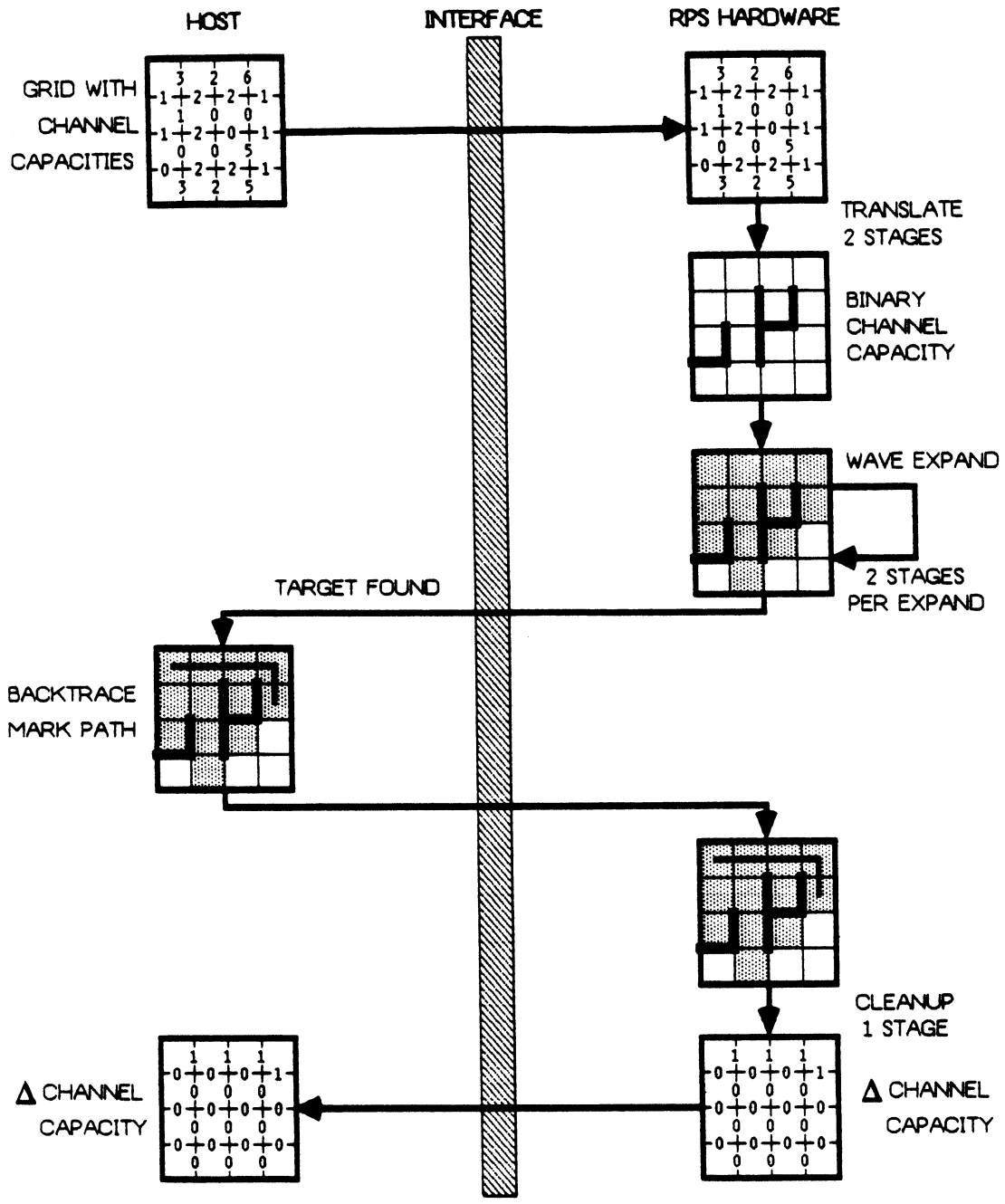


Figure 4.21 Data Flow for RPS Global Router

Phase	Implementation	Function
Binary-Capacity Translation	2 stages/translate	Mark admissible channels
Wavefront-Expand	2 stages/expand	Expand across channels
Backtrace	Host	Trace shortest path
Cleanup	1 stage/cleanup	Remove expanded cells, mark channels with fewer tracks to build Δ-capacity grid
Grid Update	Host	Update original routing grid by adding Δ-capacity grid

Table 4.4 Local Design for RPS Global Router

4.6.2. Experiments

The simplicity of the channel representation and inability to incorporate penalties for congested channels restricts the router to very simple applications. Accordingly, we have not implemented a complete router embodying these ideas, but rather, implemented and tested all the necessary stage programs. These five stage programs have all been functionally verified.

Given that the local design works, we are assured that the global design will work. Execution time can be estimated by comparison to the previous routers. The need to move the grid to the RPS hardware to begin routing, and to return the Δ-capacity grid after cleanup has essentially the same total overhead of an extra backtrace step. Similarly, the initial translation phase can be viewed as an extra cleanup step: the execution times for 2-stage translation or 1-stage cleanup are nearly identical. Recall that the two-layer router has a 2-stage cleanup, and a 3-stage wavefront expand. Hence, as a rough bound, execution time for a $2S$-stage global router is no more than twice that of a two-layer router running on a $3S$-stage pipe. Framing has less impact here, given the larger granularity of cells and the smaller 10^3 to 10^4 cell grids employed in global routing.

4.7. Generic Concerns for RPS Routers

The goal of these routing experiments is to demonstrate the feasibility and practicality of RPS-structured routing engines, and to uncover those problems that appear to be generic to all

RPS systems. Experiments with single nets verify feasibility, and support our early conjecture that performance can be improved by adding pipeline stages. Experiments with PCB's and gate-arrays, and comparison with software maze-routers verify practicality for real problems. This section discusses two design/performance issues central to an RPS-structured router. We discuss first the effects of statelessness, and examine the extent to which global data extracted by a stage can improve routing performance. Next, we present a model and rigorous analysis of the framing problem.

4.7.1. The Effects of Statelessness

The inability to extract global information from a grid streaming through a stage is a major drawback of cytocomputer architecture. Although the state-machine model is appropriate for wavefront expansion, it remains too restrictive as a general model for a routing engine. The previous experiments suggest that the following primitives be available in each stage:

- Count Marked Cells -- A count of the cells expanded by a stage is useful to determine the need for further expansions. When this count is zero, no cells are expandable and we may reframe or quit. Often, nets fail because of blocking near their pins. A small wavefront expansion from each net terminal followed by a count of expanded cells can detect this, and save the cost of trying to embed the net.
- Flag Distinguished Cells -- As a special case of the above, the ability to detect cells with special labels or at special locations is also useful to to terminate expansion. In our experiments, the host checks for target expansion, and it is usually necessary to overrun the target during expansion. Detection of target expansion in the RPS hardware appears to be more efficient.
- Measure Extremal Cells -- Examples of this include retaining the maximum cell penalty encountered, useful for path-functions with cell penalties, or determining the minimum bounding frame of the current wavefront, useful for sizing the next incremental frame. The current implementation uses worst-case framing, extending frames by the maximum distance a wavefront *might* reach, because we are unable to compute their actual extent

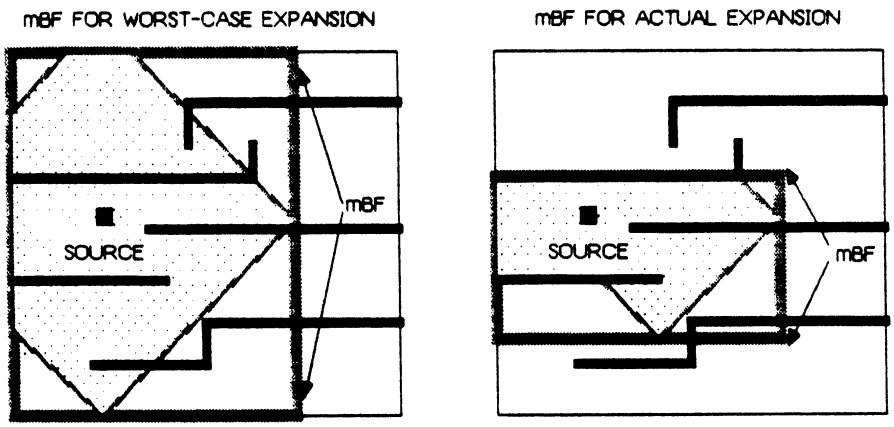


Figure 4.22 Minimum Bounding Frame after Expansion in Congested Region

using the pipeline. A source cell in a congested region may expand wavefronts that occupy an area much smaller than a source expanded in an empty grid; Fig. 4.22 illustrates such a case. This is important because minimizing the number of unreachable, unexpandable cells passing through the pipeline minimizes execution time.

Also included in the definition of statelessness is the restriction that newly computed cells are injected immediately into the output. It is of interest to pursue instead the consequences of replacing these values in the current grid to use as the basis for subsequent subarray-computations. Note first that expanding from a source cell with one pass through an S-stage pipeline now searches the area shown in Fig. 4.23. Directionality is now a factor: as the grid moves through the pipeline, subarray computations occur in the order determined by the grid raster, for example, west to east, north to south. Hence, in the figure, all cells within Manhattan radius S *and* all cells south and east of the wavefront at radius L are searched. West-to-east and north-to-south segments starting from the source are found in one pass; jogs no longer than S cells against these directions can also be found. However, the simple unit-cost expansion with source-pointing arrows no longer suffices to guarantee shortest paths. This greedy, directional expansion does not proceed in discrete wavefronts equidistant from the source.

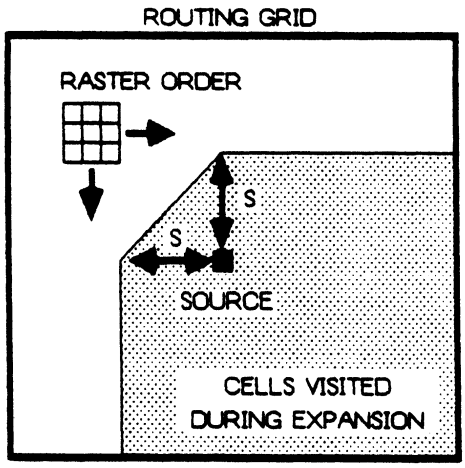


Figure 4.23 Wavefront Expansion in S-Stage Non-Stateless Pipeline

This suggests that perhaps cells should be labeled directly with their distance from the source. This succeeds, but routing time now depends on the number and direction of each net segment in the best path. In general, the first path found is not the best, and many cells are relabeled as wavefronts move around obstacles opposite to the direction of subarray computations. It may be desirable to change the raster order on subsequent pipeline passes, to allow paths to move more rapidly around obstacles. It is unclear whether this added complexity is warranted in general by the potential speedup from fewer pipeline passes.

4.7.2. Framing

Incremental framing has been shown experimentally to yield significant speedups for routers running in our RPS environment (see again Fig. 4.12). Our informal derivation, however, is unsatisfactory because it provides little information to guide the design of an RPS routing engine. Tradeoffs among pipeline speed, length, and overhead must be considered in such a design, and it is essential to compute optimal net routing times for a given hardware configuration. To compute this time, we must predict an optimal framing strategy based on the hardware configuration. Our experiments have shown that framing is significant in a short-pipe, high-overhead environment, but provide little concrete insight for other configurations. This section rigorously formulates the problem of optimal framing strategies. We develop first an

approximate algebraic model to provide some insight into the structure of the problem, then formulate a more exact dynamic-programming model.

4.7.2.1. An Approximate Model of Framing

Consider the problem of expanding all cells within a Manhattan radius L from one source cell. Using the constant frame-increment heuristic, we minimize processing time by optimizing the frame-increment f. These simplifying assumptions are made:

- Pipeline length is S stages with $S>L$. Several pipeline passes are necessary to perform L wavefront expands. One expand-step is one subarray-computation in one stage.
- Each incremental frame is square; the kth frame has edge length f_k. Successive frames extend outward a constant distance equal to the frame increment f, that is:

$$f_{k+1} - f_k = 2f \tag{4.10}$$

  As a boundary condition, the source frame is considered to be a single dimensionless point: $f_0 = 0$.
- f incremental expansions are done in each frame, where f is a multiple of S:

$$f = rS \tag{4.11}$$

- Pipeline processing time is dominated by the time to flush a grid through the pipe. Latency is ignored. The time $\tau_{pass}(r,S,X)$ to process a square $X \times X$ grid r passes through an S-stage pipeline is

$$\tau_{pass}(r,S,X) = \tau_{ovh} + rX^2\tau_{stage} \tag{4.12}$$

  which is equation (3.17) without latency and with one lumped time τ_{ovh} to account for all host overhead.
- An entire framing strategy has exactly N frames, and the total number of expand steps processed in all frames is exactly L:

$$NrS = L \tag{4.13}$$

From these assumptions and boundary conditions, frame size can be computed as:

$$f_k = f_0 + 2kf = 2krS, \quad k=1,2,...,N \tag{4.14}$$

T_{expand} is the total time to expand each frame through the pipeline:

$$T_{expand} = \sum_{k=1}^{N} (\tau_{ovh} + rf_k^2 \tau_{stage}) \tag{4.15}$$

This is expressible in terms of a single independent variable N with the substitution

$$rf_k^2 = 4k^2 r^3 S^2 = 4k^2 \frac{(L/N)^3}{S} \tag{4.16}$$

whereupon T_{expand} has closed form:

$$T_{expand} = \frac{\alpha \tau_{ovh}}{N^2} [\, \alpha^{-1} N^3 + 2N^2 + 3N + 1] \quad \text{where } \alpha = \frac{2L^3 \tau_{stage}}{3S\tau_{ovh}} \tag{4.17}$$

T_{expand} is now minimized by allowing N to become a continuous variable. Minimum time occurs at N_{opt} which satisfies $\left.\frac{dT}{dN}\right]_{N_{opt}} = 0$. For reasonable ranges of L, S, τ_{stage} and τ_{ovh}, $\alpha >> 1$ and N_{opt} can be approximated as:

$$N_{opt} \approx \sqrt{3\alpha} = \left(\frac{2L^3 \tau_{stage}}{S\tau_{ovh}} \right)^{\frac{1}{2}} \tag{4.18}$$

The optimal frame-increment f_{opt} can then be approximated as:

$$f_{opt} = \frac{L}{N_{opt}} = \left(\frac{S\tau_{ovh}}{2L\tau_{stage}} \right)^{\frac{1}{2}} \tag{4.19}$$

The form of f_{opt} satisfies our intuition about framing: when relative overhead, τ_{ovh}/τ_{stage} is large, we choose fewer, larger frames; when relative overhead is small we choose many smaller frames. By substituting N_{opt} back into the equation for processing time, we find the optimal time to be

$$\begin{aligned} T_{expand}^{opt} &= \frac{\tau_{ovh}}{3} [\, 6\alpha + 6\sqrt{3}\alpha + 1 \,] \\ &= O(\alpha) + O(\alpha^{1/2}) + O(1) \\ &= O(\alpha) = O\left(\frac{L^3}{S} \right) \end{aligned} \tag{4.20}$$

which has the expected time complexity. Framing is a practical technique to reduce the constants in (4.20) but does not alter the intrinsic time complexity.

Table 4.5 uses (4.19) to compute f_{opt} for the framing experiment of section 4.4.3. We assume $\tau_{stage} = 2\mu s$, $\tau_{ovh} = 90ms$ (approximately 3 pipeline commands) and compare with the optimum f interpolated from the experimental data in Fig. 4.12. Note that the results are in reasonable agreement, although predicted f_{opt} is uniformly too small. This is because the model uses no MBF, while the real router imposes a tight MBF and always expands fewer cells.

Pipeline Stages	Optimal Frame-Increment f_{opt}	
	Experimental	Predicted
1	$\sim$ 12-16	10
2	$\sim$ 16-20	14
3	$\sim$ 20-24	17

Table 4.5 Predicting Optimal Frame Increment

4.7.2.2. Optimal Framing by Dynamic Programming

The assumptions required to formulate the algebraic model restrict its applicability to real problems. The algebraic model assumes isotropic expansion, no maximum frame, and no pipeline latency. For long wires, all these assumptions are invalid and the model is less accurate. Moreover, the model assumes constant-increment framing, which is a practical heuristic but not necessarily an optimal solution. This section develops a dynamic programming (DP) model of framing to address these problems.

It should be noted that this model is not intended to be solved at run-time on a net-by-net basis in an RPS router. We show later that its intrinsic time complexity is too large for this use. Rather, this model is an analytical tool to produce optimal framing strategies against which practical run-time heuristics can be measured. Thus, it is useful to verify system tuning, and also to measure the sensitivity of performance measures to changes in system parameters. This section derives and solves the necessary dynamic programming equations, and examines these sensitivity issues. In Chapter VI we revisit this model and study its application to RPS system design.

We first discuss an interpretation of dynamic programming suitable for the framing problem. Fig. 4.24 shows a simple DP problem. A quantity Q is to be distributed among N stages. In stage k a decision D_k is made about how much of Q to retain; quantity Q_k enters the stage and $Q_{k-1} = t_k(D_k, Q_k)$ exits. This decision incurs a cost $C_k(D_k, Q_k)$. The task is to optimize the total cost $\sum_{k=1}^{N} C_k$ over the N decisions $D_1, D_2, \cdots, D_N$. In DP terminology, the Q_k are *state variables*, D_k are *decision variables*, $C_k(Q_k, D_k)$ are *stage-returns*, and the functional relationships $t_k(Q_k, D_k) = Q_{k-1}$ are *stage-transformations*.

To solve the framing problem, regard each DP stage as a frame[5]. The quantities distributed among frames are wavefront expansions: L expand steps are needed to find a wire of length L. The state variable in each frame is the total number of expand steps performed so far. The decision variables determine how many pipeline stages are active in each frame, and how many pipeline passes are done to process expansions. The stage-return function is the time to process wavefront expands in a frame with the pipeline specified by the decision variables. The optimization task is to minimize total processing time.

This model suffices to solve more general problems than the algebraic model. Our previous assumptions change as follows:

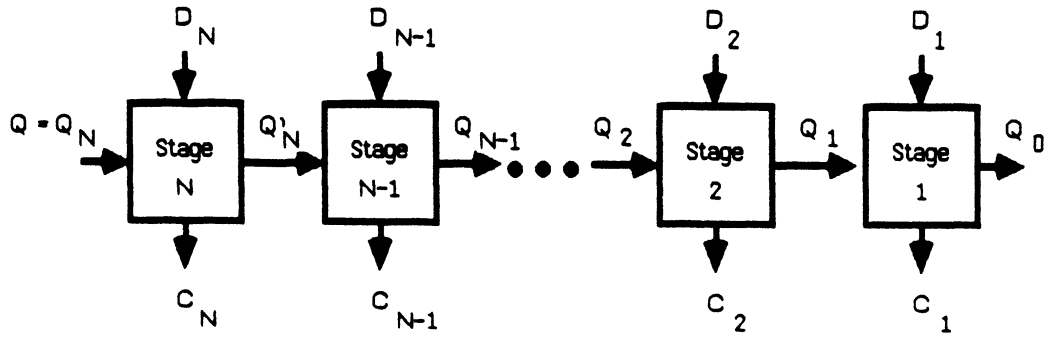


Figure 4.24 Elementary Dynamic Programming Problem

[5]It is necessary to distinguish between RPS stages, which are hardware, and DP stages, which are mathematical artifacts used to decompose the optimization problem into independent sub-problems.

- Starting from a source frame, we expand only cells that are in the specified maximum bounding frame and within a Manhattan radius L. The source frame has dimension $f_0^x \times f_0^y$. The maximum frame extends past the source frame by u,v in the horizontal and vertical directions respectively; see Fig. 4.25.
- Pipeline length is S stages, with no constraint on the relative magnitudes of L and S.
- Each incremental frame has dimensions $f_k^x \times f_k^y$. Successive frames extend outward a distance equal to the number of incremental wavefront expands performed in the stage (this is a worst case scenario as in Fig. 4.22), but no frame may ever exceed the boundaries of the MBF.
- The number of wavefront-expands done in each frame is determined by the number of pipeline passes and pipeline length in the frame: the kth frame performs r_k passes through an S_k-stage pipe. Assume one wavefront-expand is one subarray computation, so this yields $r_k S_k$ wavefront-expands.
- Pipeline processing time can be modeled arbitrarily. Let $\tau_{pass}(r, S, f^x, f^y)$ be the time to process an $f^x \times f^y$ grid r passes through an S-stage pipe. We choose a simplified version of (3.17):

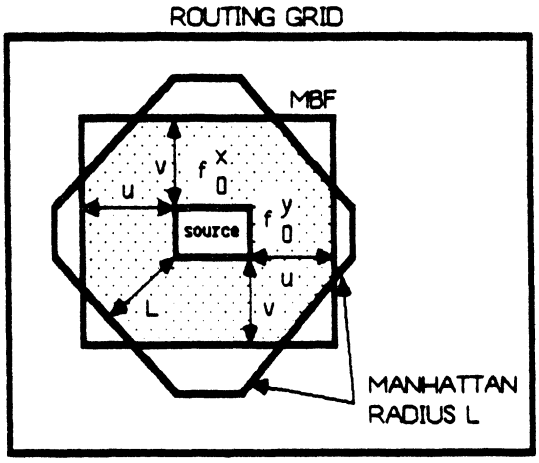


Figure 4.25 Model of Framed Routing Problem for DP Solution

$$\tau_{pass}(r, S, f^x, f^y) = \tau_{ovh} + r\left[S(f^x + 2)\tau_{stage} + f^x f^y \tau_{stage}\right] \qquad (4.21)$$

where τ_{ovh} is again a single lumped overhead time. Note pipeline latency is now included.

- An entire framing strategy has exactly N frames. An optimal framing strategy is characterized by N, and the pipeline passes and pipeline lengths per frame: r_k, S_k, $k = 1, 2, ..., N$. Other needed parameters can be computed from these.

The basic frame optimization problem then becomes:

$$\begin{aligned} \min_{N,\, r_k,\, S_k} \quad & \sum_{k=1}^{N} \tau_{pass}(r_k, S_k, f_k^x, f_k^y) \\ \text{subject to:} \quad & \sum_{k=1}^{N} r_k S_k = L \\ & 0 \le r_k \qquad k = 1, 2, ..., N \\ & 1 \le S_k \le S \quad k = 1, 2, ..., N \end{aligned} \qquad (4.22)$$

This problem can be transformed into a standard DP problem. The DP solution makes a correspondence between DP stages and frames as shown in Fig. 4.26. The kth DP stage is characterized as follows:

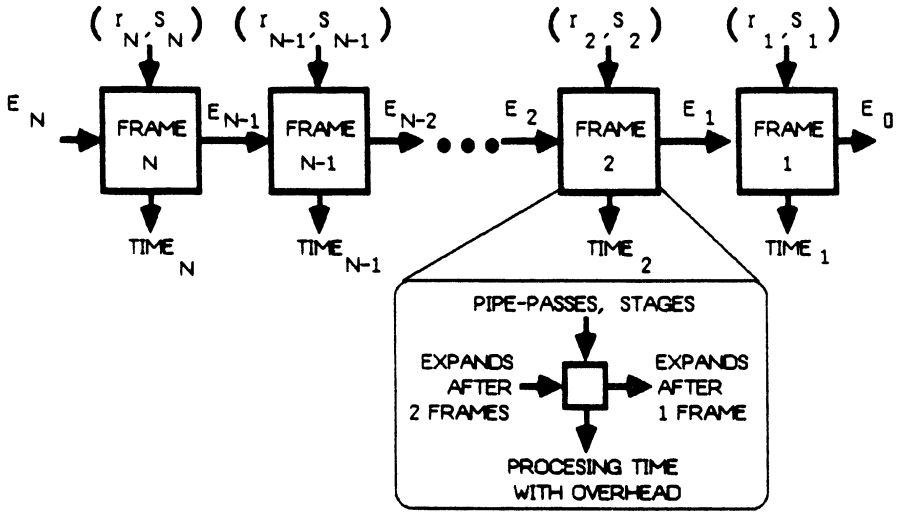


Figure 4.26 DP Reformulation of Incremental Framing

- E_k is the state variable, interpreted as the total number of wavefront-expands done after the first k frames.
- The decision variables are r_k, S_k, where r_k is the number of pipeline passes, S_k the number of active pipeline stages. Pipeline length is constant in each frame, but may vary frame to frame.
- The stage-transformation is:

$$t_k(r_k, S_k, E_k) = E_{k-1} = E_k - r_k S_k \tag{4.23}$$

that is, stage k adds $r_k S_k$ expansions to the total.

- The stage-return is the pipeline processing time $\tau_{pass}(r_k, S_k, f_k^x, f_k^y)$. We need to transform this to the form $C_k(r_k, S_k, E_k)$. Since frames increase in size with wavefront expands, but never cross the MBF, we have the constraints:

$$f_k^x = f_{k-1}^x + 2\min(u, r_k S_k), \quad f_k^y = f_{k-1}^y + 2\min(v, r_k S_k) \tag{4.24}$$

Summing over k frames gives:

$$f_k^x = f_0^x + 2\min(u, \sum_{j=1}^{k} r_j S_j), \quad f_k^y = f_0^y + 2\min(v, \sum_{j=1}^{k} r_j S_j) \tag{4.25}$$

To put the stage-return in terms only of decision and state variables, note that $E_k = \sum_{j=1}^{k} r_j S_j$. Then the stage-return is

$$C_k(r_k, S_k, E_k) = \tau_{pass}(r_k, S_k, f_0^x + 2\min(u, E_k), f_0^y + 2\min(v, E_k)) \tag{4.26}$$

which has the appropriate form.

- Given the stage-transformation, the state variables E_k are constrained as follows:

$$\begin{aligned} &E_N \geq L \\ &E_{k-1} = E_k - r_k S_k \quad k = 1, 2, \cdots, N \\ &E_0 = 0 \end{aligned} \tag{4.27}$$

The boundary conditions imply that no expansions are done prior to the first stage, and that at least L expansions are done in all. Summing over the first k state variables yields:

$$\sum_{i=0}^{k-1} E_i = \sum_{i=1}^{k} (E_i - r_i S_i) \tag{4.28}$$

which rearranges to become

$$E_k = \sum_{i=1}^{k} r_i S_i - E_0 = \sum_{i=1}^{k} r_i S_i \tag{4.29}$$

to show that E_k is indeed the number of expansions through the first k frames.

In this form, the original optimization problem can be decomposed into N sequential sub-problems. Define $T_k(E_k)$ to be the optimal time to process E_k wavefront expands in exactly k frames. Then $T_N(L)$ is the solution to the original frame optimization task (4.22). The $T_k(\bullet)$ comprise N dynamic programming equations [Nemh66] that can be solved recursively when formulated as follows:

$$\begin{aligned}
T_1(E_1) &= \min_{r_1, S_1} [\ C_1(r_1, S_1, E_1)\ ] \\
T_k(E_k) &= \min_{r_k, S_k} [\ C_k(r_k, S_k, E_k) + T_{k-1}(t_k(r_k, S_k, E_k))\ ] \quad k=2,3,...,N \\
&\text{subject to:} \quad E_N \geq L \\
&\qquad\qquad\ E_0 = 0 \\
&\qquad\qquad\ r_k \geq 0 \qquad k=1,2,...,N \\
&\qquad\qquad\ 1 \leq S_k \leq S \quad k=1,2,...,N
\end{aligned} \tag{4.30}$$

Solving recursively, we find $T_1(E_1)$, $T_2(E_2)$, $\cdots$, $T_N(E_N)$ in order. The result of solving the kth stage is $T_k(E_k)$ for all feasible E_k, and the two decision functions $D_k^r(E_k)$, $D_k^S(E_k)$, which give the optimal r_k, S_k for each feasible E_k. From these T_k and optimal decision functions, one can trace the decisions that produce the optimal framing strategy, from which the actual frames in the sequence may be reconstructed. A straightforward algorithm to find an N-frame sequence that performs exactly L expands appears in Fig. 4.27.

There are a few additional points to consider about how this model is solved practically. First, note there is no guarantee that the optimal method to find a path of length L is to perform *exactly* L wavefront expands. It may be less costly to perform a few extra expands and overrun the target than to reconfigure the pipeline. Hence, we should solve for the best framing strategy giving *at least* L wavefront-expands. Given the function $T_N(E_N)$, the best framing strategy performs L_{opt} expands, where:

$$T_N(L_{opt}) = \min_{L \leq l \leq L_{max}} [\ T_N(l)\ ] \tag{4.31}$$

In practice, L_{max} never exceeds $L + S$, and is often simply the nearest multiple of S; this accounts for the constraint $E_N \geq L$ in (4.30). Second, it appears that the DP algorithm only

Given: N, L, and description of net geometry as in Fig (4.25).

Task: Generate functions $T_k(\bullet)$ and associated optimal decision functions for N stages, then trace optimal decisions r_k^{opt}, S_k^{opt} that define optimal framing strategy.

Algorithm: Solve_Optimal_Framing

```
begin

    /* solve for T_k(•) in order */
    for k := 1 to N do
    begin
        for each feasible E_k do
        begin
            if k = 1
            then begin
                T_1(E_1) := min_{r_1,S_1} [ C_1(r_1, S_1, E_1) ]
                D_1^r(E_k) := optimal r_1 that minimizes T_1(E_1)
                D_1^S(E_k) := optimal S_1 that minimizes T_1(E_1)
            end
            else begin
                T_k(E_k) := min_{r_k,S_k} [ C_k(r_k, S_k, E_k) + T_{k-1}(t_k(r_k, S_k, E_k)) ]
                D_k^r(E_k) := optimal r_k that minimizes T_k(E_k)
                D_k^S(E_k) := optimal S_k that minimizes T_k(E_k)
            end
        end
    end

    /* trace decisions in optimal strategy */
    E := L
    for k := N downto 1 do
    begin
        r_k^opt := D_k^r(E)
        S_k^opt := D_k^S(E)
        E := t_k(r_k^opt, S_k^opt, E)
    end
end
```

Figure 4.27 DP Algorithm to Obtain Optimal Framing Strategy

solves half the problem: given a fixed N, we can find an N-frame strategy, but it appears we must re-solve for each N. A small trick alleviates this problem. If pipeline passes per frame r_k is allowed to be 0, then the DP algorithm can find a shorter strategy with essentially null, no-expand frames. Hence, by choosing N sufficiently large, we can find an optimal frame strategy with at most N frames. In practice, reasonable values of N are easily guessed.

The time complexity of this algorithm is too great for it to be solved for each net at run-time. An optimal strategy to perform exactly L expand steps in at most N stages can be obtained with time complexity $T_{DP} = O(NL^2 \log S)$, where it is assumed that the time to compute one stage-return is $O(1)$. Consider the kth DP stage. This stage examines E_k in the range 0 to L. For each feasible E_k, the stage examines decision variables r_k, S_k that satisfy the stage-transformation:

$$L \geq E_k - r_k S_k = E_{k-1} \geq 0 \tag{4.32}$$

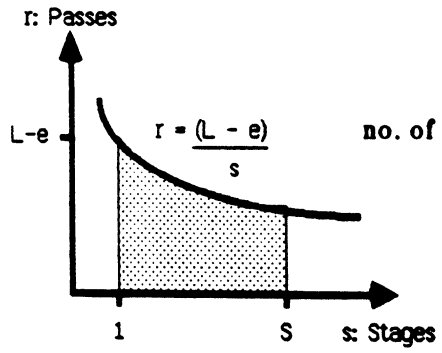
and from the feasible decision variables chooses the best for each E_k. The number of stage-returns computed is bounded by:

$$\sum_{e=1}^{L} (\text{no. of } r_k, S_k \text{ decision pairs with } r_k S_k \leq L - e) \tag{4.33}$$

Given $1 \leq S_k \leq S$, $0 \leq r_k$, the number of feasible decision pairs is the number of points under the curve $r_k S_k = L - e$ of Fig 4.28, shown there to be no more than $(L-e)\log S$. Hence T_{DP} becomes:

$$T_{DP} \approx \sum_{e=1}^{L} (L-e)\log S = \left[\frac{L^2 - L}{2}\right] \log S = O(L^2 \log S) \tag{4.34}$$

Since all N stages perform the same set of computations in order, the overall time is $O(NL^2 \log S)$. We can safely ignore the time to trace the optimal decision, as it is only $O(N)$.



no. of decision pairs $\leq \int_1^S \frac{L-e}{s}\, ds = (L-e)\log S$

Figure 4.28 Feasible Decision Variables

To verify the DP model, we recompute the expected wavefront expansion times for some of the one-layer two-point nets routed in the benchmark of Fig. 4.13. These computed times appear next to the best measured routing times in Table 4.6. Note that we should not expect an exact prediction as only expansion times are computed, the geometry of the nets is somewhat simplified (source in exact center of MBF, etc.), overhead is one lumped time, no pipeline programming times are modeled, and the real RPS router uses the constant-increment heuristic. However, we can verify that the same large-scale behavior occurs, and that the times are reasonably close. The data in Table 4.6 do exhibit the expected behavior, and are uniformly smaller than the measured times as should be expected for an optimal solution that ignores backtrace, etc.

An advantage of the DP solution is its flexibility. To obtain more accurate answers, we could modify the derivation to model the geometry of nets more accurately, and modify the stage-returns to account for more subtle timing effects. One major reformulation to consider is to recast S_k, pipeline stages per frame, from a decision variable into a state variable. The reason is that pipeline programming overhead in the current implementation is dependent on how many *extra* stages must be deactivated or activated between frames. To account for this, we must know S_k, S_{k-1} in each frame to determine $|S_k - S_{k-1}|$. Hence, S_k must be a state variable.

Net Length	Elapsed Time (sec)							
	1 Stage		3 Stage		5 Stage		7 Stage	
	Exp	DP	Exp	DP	Exp	DP	Exp	DP
16	0.533	0.126	0.467	0.108	0.516	0.099	0.450	0.099
32	0.767	0.302	0.566	0.165	0.650	0.140	0.516	0.127
64	1.68	0.829	0.950	0.390	0.900	0.271	0.734	0.233
128	5.33	3.50	2.30	1.37	1.78	0.913	1.47	0.719
256	25.1	19.3	9.40	6.89	6.38	4.32	4.87	3.22
512	156	118	54.9	40.5	34.5	24.7	25.4	17.9

Table 4.6 Comparison of DP Routing Times with One-Layer Benchmark

Given the ability to predict optimal expansion times over a wide range of system configurations, we can now obtain a qualitative picture of the sensitivity of our performance metrics to variations in system parameters. As an experiment, we vary net length L, pipeline stages S, and times τ_{ovh}, τ_{stage} as follows:

L	=	32, 256 cells (orthogonal 2-point nets from Fig. 4.13)
S	=	1, 2, 4, 8, 16, 32, 64, 128 stages
τ_{ovh}	=	1, 10^2, 10^4 time-units
τ_{stage}	=	1 time-unit

This experiment varies wire length, pipeline length, and relative overhead τ_{ovh}/τ_{stage} over wide ranges of feasible values. For each of the 48 unique problem/hardware configurations generated, we solve for the optimal framing strategy. Fig. 4.29 plots the results. For each wire length, three plots are given:

- Expansion Time vs. Pipeline Length -- Expansion time is an accurate indicator of total routing time; $\log_{10}$ time versus $\log_2$ stages is displayed, parameterized by relative overhead τ_{ovh}/τ_{stage}.
- Frames vs. Pipeline Length -- As an indicator of the structure of an optimal strategy, we plot the number of frames in the optimal framing sequence versus $\log_2$ stages, parameterized by relative overhead τ_{ovh}/τ_{stage}. Note than in the optimal sequence, the frame-increment (expands per frame) is no longer constant.
- Ratio of Naive-Time/Optimal-Time vs. Pipeline Length -- To determine whether incremental framing is actually worthwhile, define the *naive* time to be the time to process the entire MBF of the net (no framing) through the pipeline for expansion. The ratio of naive time to optimal time is shown versus $\log_2$ stages, parameterized by relative overhead τ_{ovh}/τ_{stage}.

These results provide several insights. High overhead is more detrimental to short nets, but has little effect on long nets. However, overhead significantly influences the structure of the best framing strategy; that is, although the optimal times may vary little, the manner in which we arrive at these times varies greatly with overhead. Savings due to framing are more significant for short pipelines, and decrease with added stages. With low overhead, long complex framing sequences are justified; with high overhead, short sequences prevail.

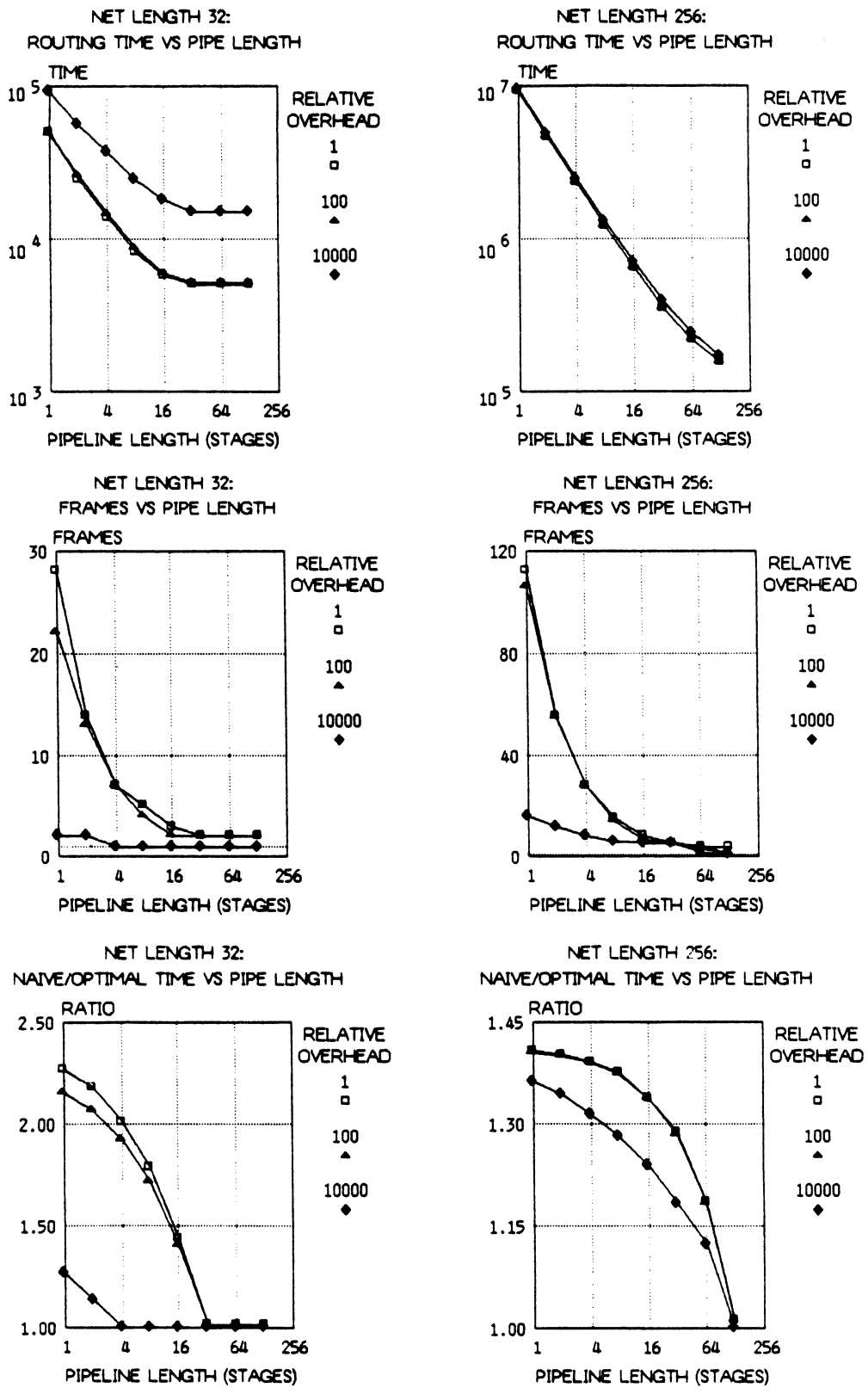


Figure 4.29 Sensitivity Studies for RPS Routers

4.8. Connections to Picture-Processing Revisited

To conclude our examination of RPS routers, we outline briefly how the image-operators for which the current hardware was originally designed may be applied to routing problems. Although the experiments in this chapter clearly demonstrate the need for a subarray stage optimized for routing tasks, some of the image-operators appear to be useful. For example, a routing grid with embedded penalties can be viewed as a grey-level image where each cell's penalty value is its grey-value. The operation of growing regions of enhanced or diminished cost around objects in the grid can be conveniently interpreted as an application of the grey-level operators. For example, consider how to mark all cells within radius R of a pin with a cost determined by their distance from the pin. A cell at radius ρ is to receive cost R/ρ. Viewed as a 3-dimensional surface, this cost-surface around a pin is a section of a cone. From an operator viewpoint, this is dilation of a point by a cone as in [Ster80a], which can be performed on the current hardware. [NHLV82] describes a method to reduce dependence on wire ordering by growing additive regions of cost around pins; this operation turns out to be precisely a grey-level dilation as defined in Sec 3.2.2. The via exclusion phase of cleanup for the two-layer RPS router is actually implemented as an elementary dilation. As another application, consider how one might route wires wider than one single cell. We cannot simply route the wire as usual and then increase its width, since it might overlap other wires or violate design rules. Instead, mark as *free* only those cells at which the wider wire could be centered. In morphological terms, this is an erosion operation. On this new grid we route as usual, assured that the final path will violate no design rules. After backtrace, we dilate out the new unit-width path to mark cells that become obstacles due to the presence of this wider wire. In general, the image-operators find some use for pre- and post-processing of routing grids.

4.9. Summary

This chapter studies RPS-structured maze-routers. Maze routing is reviewed and an analysis of the intrinsic complexity of software, array and RPS routers is presented. Functional one-layer and two-layer RPS routers are designed and analyzed in our prototype environment, and a workable scheme for global routing is suggested. PCB's and gate-arrays have been

successfully routed. These routers verify our original conjecture that RPS-structured machines can be fast, practical routing engines. Moreover, existing RPS routers, including our prototypes, can be improved incrementally and gracefully: additional pipeline stages improve execution time; additional cell memory enables larger problems to be tackled directly. Experiments over a range of pipeline lengths, and extrapolations based on experimental measurements support this claim. Critical design issues such as statelessness, and performance issues such as incremental framing, are identified and analyzed. Approximate and exact solutions to the optimal framing problem are rigorously derived and applied. Using these solutions, we study the sensitivity of routing performance to changes in the hardware environment. The success of these systems and experiments argues for the design of an RPS architecture optimized specifically for routing.

CHAPTER V

DESIGN RULE CHECKING IN AN RPS ENVIRONMENT

5.1. Introduction

This chapter examines design rule checking in an RPS environment. We discuss this application much more briefly than maze-routing for three principal reasons. First, DRC is a simpler, more obvious application for RPS machines, given their antecedents: the analogy with picture-processing is much stronger for DRC than for maze-routing. Second, concurrently with the original DRC work by Mudge *et al.* [MuLT81a, MuLT81b], and our subsequent work [MRLA82], implementation of cellular DRC algorithms on different machine architectures was studied in [BlSv81, Blan82, Seil81, Seil82]. Given this body of research, we need spend less time justifying the basic feasibility of the approach. Third, based on discussion with us, Blank has compared the performance of a cytocomputer DRC with other DRC engines [Blan82]. Hence, we need not reformulate these comparisons.

Accordingly, the first section reviews DRC in general, and examines how cellular DRCs and DRC engines fit into the spectrum of DRC tools. The next section discusses the basic structure of rules checking on an RPS machine, and reviews some of the differences between DRC and traditional picture-processing. We then look at some simple examples of how the image operators of Sec. 3.2.2 can be used to specify a DRC. We construct an experimental implementation of one specification to verify its correctness, and examine its performance. Finally, we suggest some alternative IC mask operations for an RPS-structured machine.

5.2. DRC Reviewed

The design rules for an IC mask are a set of geometrical constraints on mask features[1] imposed by the the wafer fabrication process. The two general approaches to the implementation of DRCs reflect the data-structure chosen for the IC mask. Geometric-shapes checkers perform checks on masks represented as sets of intersecting polygons or rectangles [Bair78]. Corner-stitched rectangles are a popular variant of this idea [Oust84]. Grid-based checkers work with a mask represented as a grid whose cells are labeled according to the presence or absence of particular mask layers. Both nonuniform grids, in which the chips are dissected into contiguous rectangles of arbitrary size [LoTh79], and uniform grids whose cells are unit squares have been used. For uniform grids, raster-scan approaches [Bake80, EuMu82] have been developed that access a grid in raster order and check *local* design rules. The idea here is to pass a small window over the grid and identify the local violation-patterns appearing in the window. This latter approach, which we can easily identify with subarray-computations, motivates a DRC on RPS hardware.

Roughly speaking, a design rule checker performs three functions on mask features: connectivity resolution, layer combination, and tolerance checking. Connectivity resolution merges discrete shapes on the same layer into a single larger shape if they overlap; connectivity is similarly assessed across several layers, e.g., across contact windows. Layer combination creates new layers from Boolean combinations of existing layers, e.g., the intersection of several layers. Tolerance checks determine whether a local group of shapes on one or more layers satisfies some spatial constraint: corner/edge separation, incursion, inclusion, exclusion, size, area, perimeter.

Different mask data structures have different advantages and disadvantages with respect to these three DRC operations. The singular advantage of cellular grids is, unsurprisingly, their uniformity: processing is homogeneous across the grid. This is particularly attractive for a hardware-based DRC. For masks represented as grids, local connectivity and layer combination are easily computed. Overlapping shapes automatically become a single entity as the cells

[1]The following terms are interchangeable: *mask, mask features, artwork, layout, geometry.*

within the shapes are labeled as belonging to a particular layer, and Boolean combinations performed globally across several layers are simply performed on each cell in the mask. Tolerance checks are more interesting since they require not just cell by cell processing but also pattern recognition operations on spatially distributed groups of cells. We concentrate on these in subsequent sections.

There are also several serious disadvantages to uniform cellular grids. Six basic problems are: global connectivity resolution, connectivity pathologies, oblique geometry, restricted feature sizes, increased rule complexity with distance, and mask size. We treat each separately.

- **Global Connectivity Resolution**

  Features widely separated on a mask may be connected electrically. Such global connectivity is difficult to resolve if we are restricted to strictly local processing of grid cells. Resolution requires propagating nodal information globally around the grid. It is not impossible to perform this on a cellular grid if some global processing is available [Bake80], but it is expensive to maintain a node-number in each grid cell.

- **Connectivity Pathologies**

  Because we may not be able to identify electrically connected features, we may flag spurious rule violations, e.g., features that appear too close together may not violate design rules if they are electrically equivalent.

- **Oblique Geometry**

  Layouts represented with polygon-based data structures may contain features of arbitrary shape and arbitrary size. Grid-based representations are generally restricted to orthogonal artwork only.

- **Restricted Feature Sizes**

  Layout represented on a grid must conform to the size of each grid cell. Mask features are restricted to multiples of unit the cell size. In particular, the design rules must also conform to this restriction. This is the form of the Mead and Conway NMOS rules [MeCo80].

- **Increased Rule Complexity with Distance**

  In a polygon-based layout representation, sizes are encoded directly in each feature. It is no more complex to check a design rule involving a distance D_1 than to check with a different distance D_2. For a cellular grid, complexity increases with the critical sizes that parameterize design rules. For example, a width-W check at a point may require visiting all cells within a radius-W circle. Rule checks involving large distances, e.g., verification of minimum size for a bonding pad, thus consume more time than for a small distance.

- **Mask Size**

  Individual grid cells must be small to resolve all features of a mask and the associated design rules. Consequently, mask grids can be sizable. Consider a large chip drawn on a cellular grid. If f is the minimum feature size of the given layout, some commercial microprocessors drawn on a grid of $f \times f$ squares require 10 to 15 million cells [FrSp81]; for reference, a 300 mil $\times$ 300 mil chip drawn on a 1μm grid has about 60 million cells. The practical problem is how to generate, rasterize, or store layouts with grids this large.

Because of these problems, some advocates of the polygon approach have dismissed cellular approaches as inadequate [SzVW83]. We prefer to view cellular DRC, as implemented in hardware, as embodying a tradeoff between quality and speed. That is, we do not suggest that a cellular DRC engine is a replacement for a traditional polygon DRC. Rather, we suggest that it is a separate point in the spectrum of DRC tools. Polygon checkers provide a very complete rule check, but are often slow; run times are often measured in days on a mainframe [BHML82]. On the other hand, a cellular DRC engine is potentially quite fast, but sacrifices some quality. For that large class of designs for which grids are an appropriate representation, we argue that this type of rules check, executed rapidly in hardware, is useful as feedback during the design process.

Note that not all these problems associated with cellular checkers are insoluble. For example, some oblique geometry is representable at the cost of increased storage or processing complexity. [Seil82] discusses a scheme for 45° artwork based upon conditional labeling of cells during rasterization. [LoTh79] suggests a scheme using extra storage for each cell; we

look at how formal image operators might be extended to this scheme in a later section. Inclusion of oblique geometry is a classical tradeoff between designs and tools: chips that permit obliques are always denser; tools that restrict to orthogonal features are always faster. It has been argued that obliques are a questionable luxury that may become too expensive to check in the face of VLSI complexity [Losl80]. Similarly, restriction to layouts on a unit-grid appears not to be untenable. The popularity of the simplified NMOS design rules of Mead and Conway [MeCo80] and their migration to other technologies [Lyon81, Gris82] support this claim.

The problem posed by the size of realistic layouts is also addressable. Given the bias of this thesis toward hardware solutions, we suggest that the problem of digitizing the layout be addressed in hardware, along with the actual rules checking mechanisms. For example, we might simply dedicate a large disk and cell buffer to an RPS-based DRC system. The layout need only be rasterized once and stored to be later streamed through the RPS pipeline. We need not store the grid in flattened form here. If pre- or post-processing is available at the ends of the pipeline, a simple run-length encoding would reduce storage requirements, and hence data transfer time. More complex bit-map encoding schemes, e.g., [Wilm80], might be possible with extra hardware. [Seil82] discusses a single-chip polygon-to-raster converter to supply the cell stream to a cellular DRC engine. This system is apparently functional; see [Seil84]. Hence, we conclude that some combination of memory and rasterization hardware suffices to manage the size problem.

5.3. Elementary DRC in an RPS Environment

We assume that individual masks are represented as bit-planes. A grid with b-bit cells can thus store b different masks representing the input layout, intermediate results, and final output. This output is one or more bit-planes of error masks giving the location of specific rule violations. If space is available, we may wish to produce several error masks for different types of violations; if not, we may lump all errors into a single mask.

A DRC algorithm is a sequence of subarray-computations that checks all the individual design rules. For each rule we have a short sequence of subarray-computations to identify locations of violations; it is assumed that all rules are local in extent. Rules whose local extent fits

within the size of the subarray in each stage may be checked in one subarray-computation: one template-match suffices. Rules with larger critical distances require several subarray-computations, each of which produces an intermediate result. Hence, it is useful to have cells wide enough to provide several of these temporary bit-planes. A complete DRC is the concatenation of these individual sequences of subarray-computations. If individual cells are wide enough to provide all the necessary intermediate and error bit-planes, a K-step DRC can be performed in one pass of the grid through a K-stage pipeline. If pipeline length S is less than K, we require $\lceil K/S \rceil$ passes, and must store or regenerate the grid for each pass.

There are two basic advantages of the RPS organization for DRC. First, a long pipeline performs many individual rule checks in parallel. Processing time for an $X \times Y$ grid is the pipeline flush time, $XY\tau_{stage}$, plus the pipeline latency which is small in comparison for large grids. Second, we can accommodate large grids directly, without any sectioning or edge effects: with line buffers of length L in each stage, we can handle any $L \times Y$ grid directly. In addition, with programmable stages, we can perform a variety of different DRCs.

Note that DRC tasks resemble some typical geometric picture-processing tasks, given the focus on local processing and pattern-matching. Indeed, central features of the RPS organization (parallelism, accommodation of large grids) evolved in picture-processing applications. There are, however, some important differences to note. IC masks are synthetic artifacts; there are no errors introduced in the translation from original format into a grid.. In contrast, ordinary images are typically acquired over noisy channels, and hence, much of their processing concerns the identification and separation of the data from the noise.

Masks, because of their extreme size, are digitized at the minimum acceptable resolution. They are characterized as having mostly very small features. Ordinary images are typically acquired at greater resolution to avoid noise problems: a difference of a few cells in one feature is insignificant. Differences in resolution imply differences in processing. If we apply the formal image operators of Sec. 3.2.2 to an ordinary image, it is safe to make reasonable approximations for the geometric operations of interest, e.g., eroding or dilating with a disk. This is not true for IC masks, where much of the critical processing occurs at distances of only a few cell widths; we must be extremely careful in our definition of geometric operations here.

Finally, note that ordinary acquired images are typically, but not always, much smaller than an IC mask.

5.4. Local Design of Simple Tolerance Checks

An adequate DRC may be constructed by employing only layer combinations and some simple tolerance checks. In particular, many rules can be expressed with width or spacing checks. Since layer combination is a trivial scalar operation on each cell, we focus on width and spacing checks.

Note first that we require only one of these two checks, e.g., a spacing check is a width check on the complement of a mask. All tolerance checks in a cellular DRC have the same intuitive basis: two features fail a spacing check if, after *expanding* them, they touch; a feature fails a width check if, after *shrinking* it, it disappears. Our intention in this section is to study how to recast this intuition into the formalism of the image operators of Sec. 3.2.2. The advantages of this are twofold. First, we are provided with a compact notation for writing rule-checking algorithms. We wish to avoid the exhaustive pattern specification that often characterizes template-match operations. Second, we are provided with some formal rules for manipulating these algorithms, such as knowing which operations commute or associate. This is the methodology we suggested in [MRLA82, RuMA84].

To illustrate this idea, consider a width-W check to identify mask features less than W-cells in diameter. The algorithm is based on the simple observation illustrated in Fig. 5.1, which shows a mask on which a width-W check is to be done. Slide a disk of diameter W around inside the mask to all possible locations at which it may be completely contained (Fig. 5.1b). While it slides, mark those points covered by the disk and trace the path of its center. It is clear the disk should not pass through regions which are too narrow, i.e., regions which fail the width test. Except for some square-corner effects, those regions left uncovered all violate the width test (Fig. 5.1c). Note also that the region traced out by the center of the disk is not connected across the diagonal neck of the mask.

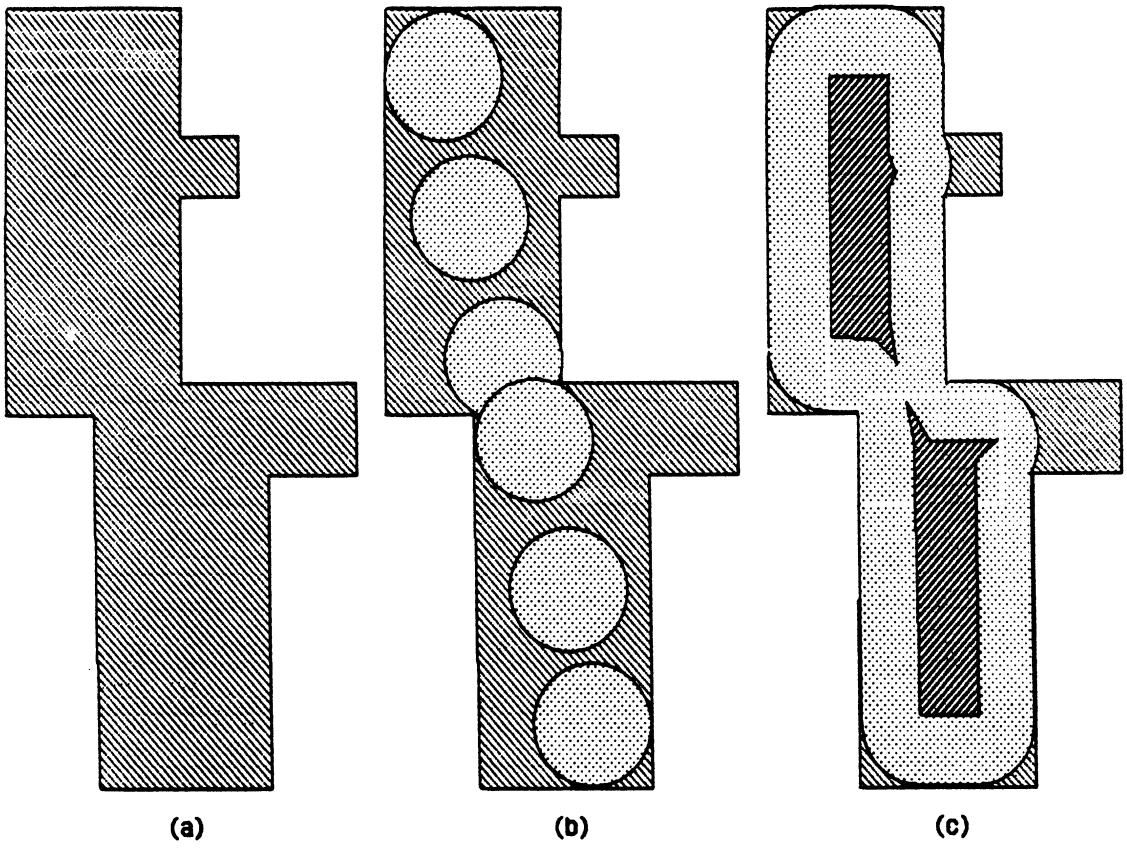


Figure 5.1 Basis for Width Checking Algorithm

With these observations we can construct an executable algorithm. Fig. 5.2 defines two geometric figures useful for this algorithm: $S(r)$ is a square with edge length r cells, and $QC(r, q)$ is a quarter-circle of radius r in quadrant q. Because we are using orthogonal masks, we abandon the diameter-W disk in favor a $W \times W$ square. The operations of Fig. 5.1 all have a formal interpretation: sliding the disk around inside a mask feature is an *opening*; tracing the center is an *erosion*; searching for disconnected components of the center can be accomplished by a *conditional dilation*. This illustrates some of the geometric intuition permitted by these operators. Fig. 5.3 gives the algorithm precisely and illustrates it geometrically. This particular algorithm tags any region smaller than $W \times W$ cells, and tags the *north* side of each diagonal pinched-neck width violation. Other schemes to indicate errors are possible, as are other geometric interpretations for width-checking; see [MRLA82].

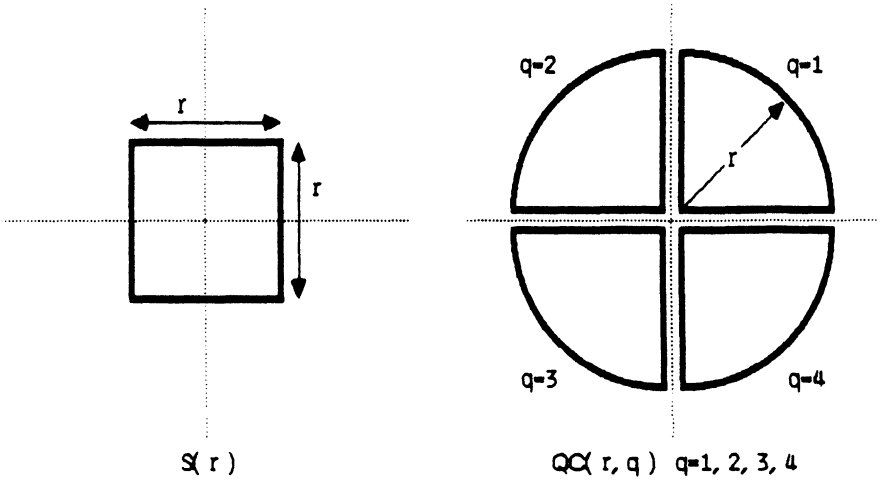


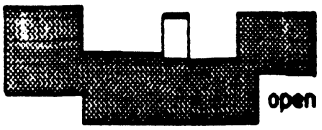
Figure 5.2 Geometric Figures for Width Check Algorithm

The only question remaining is how to realize the geometric shapes in Fig. 5.2. Recall from the discussion of Sec. 3.2.2 that we typically formulate algorithms using general geometric shapes and then decompose them into elementary subarray-computations using the rules from the algebra. It is easy to build the square $S(r)$ by successively dilating smaller squares. The quarter-circle $QC(r, q)$ is more interesting. [Blan82, Seil82] effectively build circles for their DRCs using successive 4-way and 8-way expands from a point, as shown in Fig. 5.4. The appropriate number of expands by each shape provides an approximation to a circle which can be optimized empirically. We consider an alternative interpretation. First, recognize that expands are really dilations, and that a 4-way expand is a digital approximation to dilation by a diamond-shaped quadrilateral, and that an 8-way expand is likewise a dilation by a square (see Fig. 5.4) Since dilation commutes and associates, we may dilate together all the diamonds first, and then all the squares. Since these are both convex shapes, the individual dilations simply *scale* the original figure, e.g., dilating p diamonds produces a diamond p-times larger. It is then easy to see that the resulting continuous shape has the dimensions shown in Fig. 5.4. With this knowledge, elementary calculus suffices to minimize the area between this approximation and a perfect circle. Thus, we can determine directly the optimum sequence of elementary dilations to form a circle, and hence also to form $QC(r, q)$.

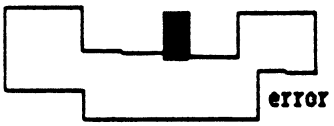
Given: A mask, "M", represented as a bit-plane.

Task: Produce another bit-plane, "error", indicating the locations of violations for a width-W rule check.

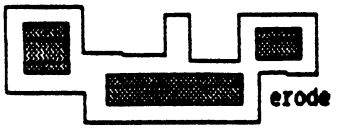
Algorithm: Width_Check



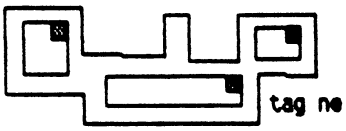

```
/* find simple orthogonal errors by opening mask M */
opened_M := M_S(W)
```



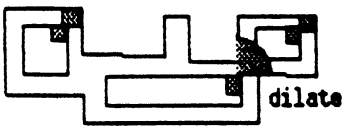

```
/* anything not in the opening is an error */
error := error ∪ (M - opened_M)
```



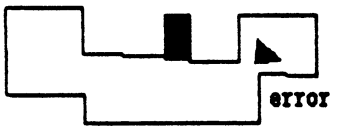

```
/* trace center of S(W) sliding in M by eroding it.
** (Note we actually have this erosion from
** the opening above; we could save and reuse it)
*/
Center := M ⊖ S(W)
```



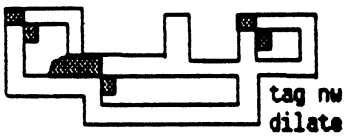

```
/* tag northeast corners of center. this prepares
** to identify breaks in Center which restrict S(W)
*/
NE_tag := match( center cells without north, northeast,
                 or east neighbors in Center )
```



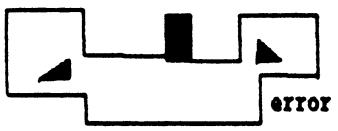

```
/* search northeast within radius W of tagged
** corners, but only cells that are in M.
*/
NE_break := (NE_tag ⊕ QC(W,1)) ∩ M
```




```
/* where break hits a center component
** (but not a tagged corner) is a violation
*/
error := error ∪ (NE_break - NE_tag)
```




```
/* same steps to the northwest for other diagonal error */
NW_tag := match( center cells without north, northwest,
                 or west neighbors in Center )
NW_break := (NW_tag ⊕ QC(W,1)) ∩ M
error := error ∪ (NW_break - NW_tag)
```




```
/* final result is in error */
```

Figure 5.3 Width Check Algorithm

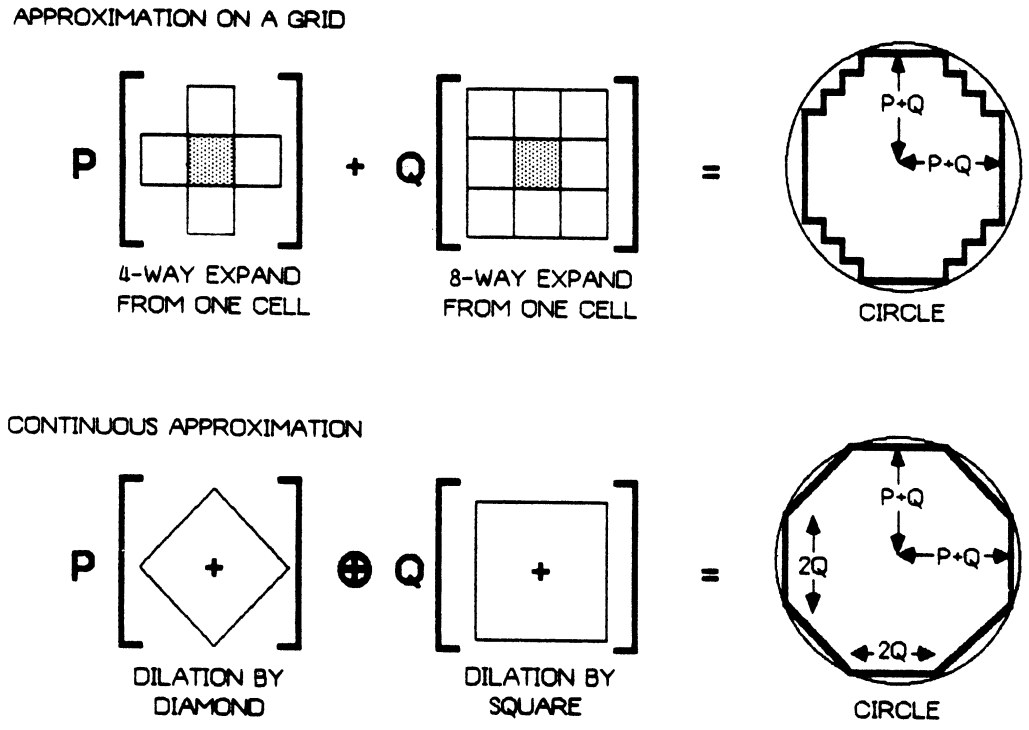


Figure 5.4 Approximating Circles an a Grid

In our experience, this approach has proven to be a particularly useful way of viewing the geometric operations in each rule check, and to provide a compact notation for writing down the algorithms.

5.5. DRC Experiments

We conduct a few simple experiments to verify our interpretation of the width check algorithm. A 3-cell width check is implemented in our RPS environment. This is the smallest non-trivial test requiring more than one subarray-computation: it requires eight subarray-computations. First, we define a simple 64 $\times$ 64 test mask and check it. The result is shown in Fig. 5.5. Total processing time is about 0.5 seconds elapsed for a 3-stage pipeline. All errors are correctly flagged. As a more interesting experiment, we stack this test mask upon itself vertically 64 times to yield a much larger 64 $\times$ 4096 mask. This is intended to represent one

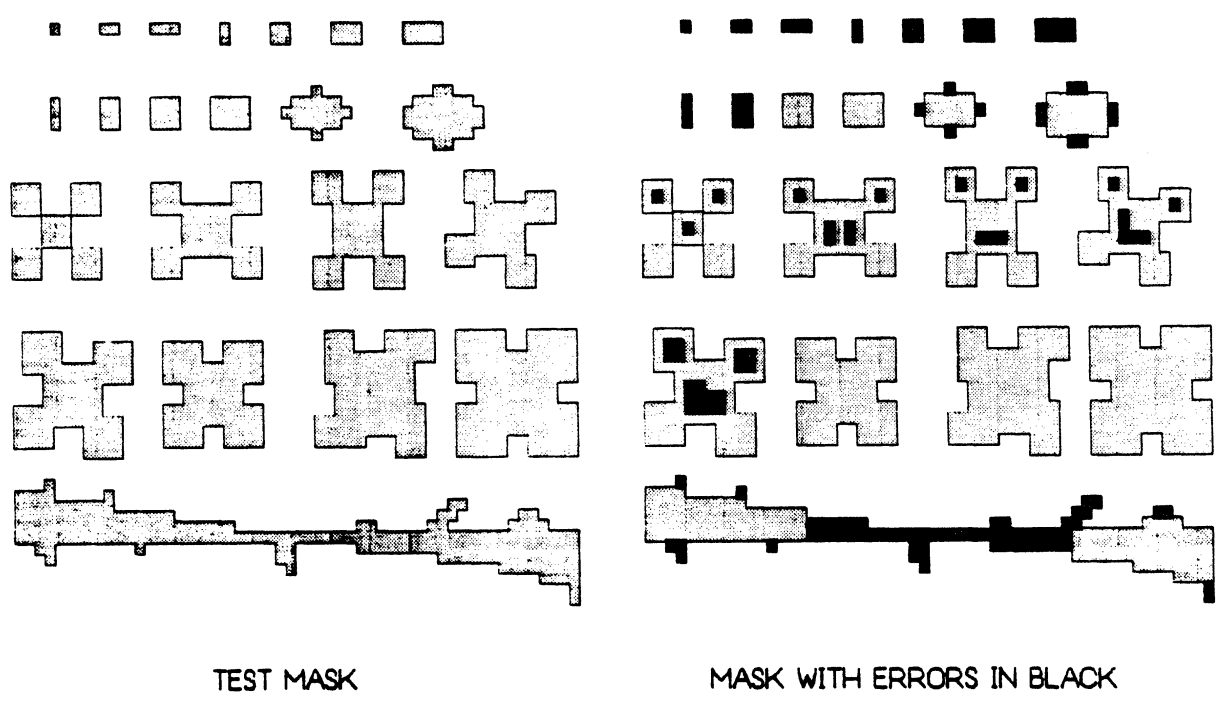


Figure 5.5 64 × 64 Test Cell and Width Checked Result

strip[2] of a chip of realistic size. The width check is then performed on this strip for various pipeline lengths. The global design for all these experiments is trivial: we digitize the mask off-line, transfer it to the cell-buffer, and perform the necessary pipeline processing, and return the grid. (Schemes appropriate for realistic masks have been discussed in Sec. 5.2.) Table 5.1 gives the resulting elapsed, host CPU, and IO times. Note that elapsed pipeline processing time decreases again as $1/S$. IO overhead time is significant for moving grids of this size; without careful design and some hardware support it is likely to be the major performance bottleneck.

Depending on the sophistication of the checking needed and extra processing required to put meaningful labels on errors, we estimate that a DRC for the Mead and Conway rules will take between 150 and 250 subarray-computations. Table 5.2 estimates the time to run such a DRC on a 4096λ × 4096λ chip for different pipeline lengths. We assume processing is done in

[2]Note that if strips are to be used, it is always best to employ vertical strips: this minimizes pipeline latency,

Pipeline Length	Host Load Factor	Strip IO Time (sec)		Pipeline DRC Process. Time (sec)		Total Time (sec)	
		Elapsed	CPU	Elapsed	CPU	Elapsed	CPU
1	2.26	2.849	.200	6.089	.367	8.933	.567
2	1.78	3.532	.283	3.304	.150	6.566	.434
3	1.62	2.750	.200	2.350	.200	5.100	.400

Table 5.1 DRC Strip Experiments

contiguous vertical strips, each 64 cells wide and overlapped by 4 cells to avoid errors, and sum all strip times. We use the transfer time for the 3-stage single strip test (Table. 5.1), and assume that the host generates these strips as fast as the pipeline requires, i.e., with negligible delay between strips. Pipeline processing time is estimated is estimated via (3.17), where the pipeline overhead times are taken to be dominated by the grid transfer time. As expected, processing time improves with additional stages, and approaches an asymptote determined by the IO times.

[Blan82] has formulated relative performance comparisons among the DRC engines of [BlSv81, Seil82] and cytocomputers, and the software rule checkers of [Bake80, NoNM81]. A wide range of DRC tasks is considered. Cytocomputers compare quite favorably to the software implementations and to other hardware; see [Blan82] for details of this study.

Pipeline Length	Estimated DRC Time (sec) for 4096λ × 4096λ Mask	
	150-Step-DRC	250-Step-DRC
1	5621.	9239.
10	737.2	1099.
50	305.8	380.9
100	266.9	304.0
250	230.7	231.7

Table 5.2 Estimated DRC Times for 4096 × 4096 Mask

which depends on the width of the strip.

5.6. Extensions to Alternative Mask Operations

This section suggests two extensions related to RPS-based DRC implementations. We look first at a method to accommodate 45° mask geometry, and then examine the use of RPS hardware as an interactive geometric query processor.

One criticism of raster DRC approaches is that they are generally restricted to orthogonal artwork. [LoTh79] suggests a fixed grid structure which admits 45° lines. The basic idea is to to draw an X through each cell, thus replacing each square cell with four small triangles, and to use four bits to represent the four regions produced. This structure is appealing but has some inherent problems. First, shrinks and expands are more complex; we must actually trace object boundaries to correctly determine where the new edges fall. Second, a "bit-plane" is now a more complex object requiring four times as much storage. Third, it is not immediately clear how any image operators fit in here.

These problems arise because this grid is not as simple and as symmetrical as a square tessellation. We propose an alternate interpretation of this grid. Instead of viewing this as a square grid each of whose cells is cut into four pieces, we will regard this as two separate, superimposed grids as shown in Fig. 5.6. Each grid is an ordinary square grid, except one has an orthogonal orientation, and the other has a diagonal orientation and is smaller than the first.

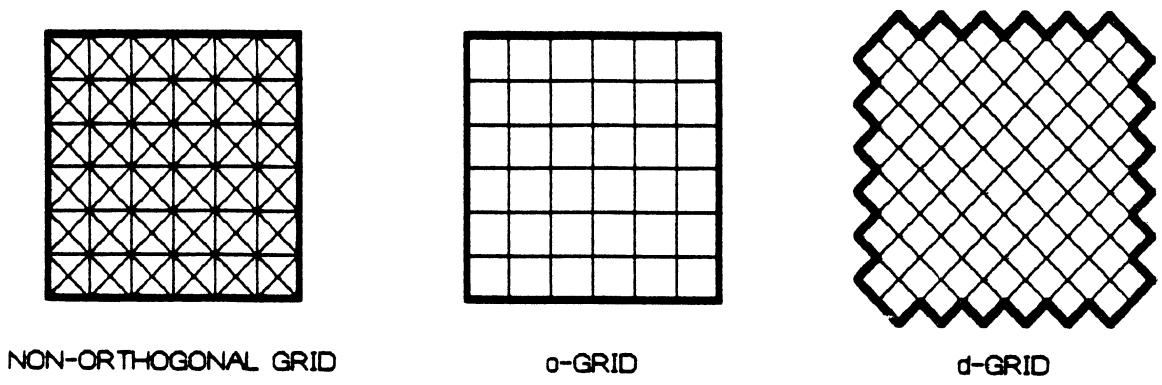


Figure 5.6 Alternative Non-Orthogonal Grid Interpretation

Call the first the *o-grid*, and the second the *d-grid*. Note that the ratio of cell sizes here is $1 : 1/\sqrt{2}$.

Let X be a figure with orthogonal and diagonal edges. Represent it in this new structure by a pair of figures (X_o, X_d) where X_o is on the o-grid and X_d is on the d-grid. It is necessary that X be recoverable from X_o and X_d, but it would be convenient if X could be manipulated solely in terms of these two, so that we never need actually recreate X. The obvious mechanism is to define X as some set-theoretic combination of the plane areas represented by the cells in the two grids. Fig. 5.7 shows an example in which $X = X_o \bigcap X_d$.

Several questions suggest themselves concerning this proposed representation scheme:

- Does this scheme require any less than the four bits/cell of the original approach?
- How can this scheme be represented physically?
- Is this scheme as flexible as the original approach?
- How do formal operators work here?

In answer to the first question, it is easy to see that we need only three bits/cell. If we consider an $N \times N$ o-grid and all the cells in the d-grid touching it, we find there are $3N^2+2N$ total cells covered, and hence $3N^2+2N$ bits required. For reasonably large N, and ignoring the

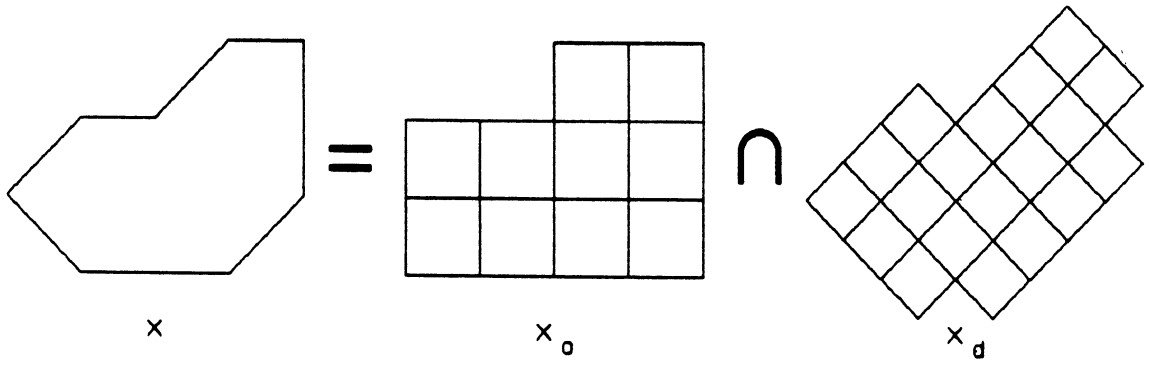


Figure 5.7 Example Representation on o-Grid/d-Grid

edges, we require essentially three bits per o-cell, giving us a savings of about 25%.

This immediately suggests an answer to the question of physical representation: with each o-cell associate two d-cells, so that each o-cell stores three bits. If we choose the north and west adjacent d-cells for a given o-cell, an o-grid with three bits per cell can represent the tessellation of Fig. 5.6.

Because the original approach uses $4N^2$ bits and we use fewer, we immediately know that this scheme cannot be as flexible as the original. Assuming that X is defined as $X_o \bigcap X_d$, what is lost is the ability to represent 45° notches in solid figures. For example, if we tried to represent Fig. 5.8a , we would actually get Fig. 5.8b. Similar problems occur for other combinations of o-grid and d-grid.[3]

The answer to how the formal image operators apply here is more complex. Ideally, given some binary morphological operator f on a standard square grid, and two figures $X = (X_o, X_d)$ and $Y = (Y_o, Y_d)$, we would like to define F, the extension of f to the o- and d- grids, to be

$$F(X, Y) = (f(X_o, Y_o), f(X_d, Y_d)) \tag{5.1}$$

to decouple completely the o-grid and d-grid computations. Unfortunately, this turns out not to work. Assuming again that we intersect o-grid and d-grid, Fig. 5.9 shows a counter example. Shape S fits into shape X in just one way, and so we would expect to have the opening $X_S = S$.

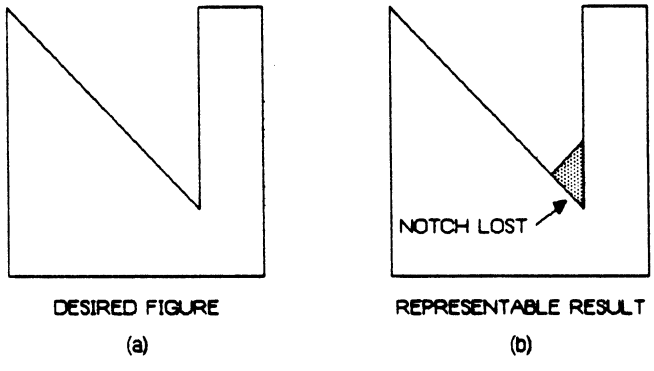


Figure 5.8 Unrepresentable Oblique Notches

[3]Such small, sharp features would likely be lost during the processing of the wafer anyway. Hence, this appears not to be a serious problem.

An incorrect result is obtained because of the *registration* problem. Operating with S on X, S_o and S_d are not independent: they have a fixed relationship, one above the other. In this example, S_o fits inside X in two places, but S_d fits in precisely one. This violates our general expectation for how S should behave; we cannot have part of it fit and part of it not. Although the nature of the difficulty is understood and seems correctable, we have no formal characterization of how to deal with it.

As a second extension, note that the individual geometric operations comprising a DRC might also be applied elsewhere. In particular, a programmable RPS engine can perform geometric queries on a digitized layout, with the query language based on the formalism of the image operators. For example, the query *"find all rectangular transistors with aspect ratios no greater than 7 and within distance 10 of of a vertical metal bus"* is a complex but well-defined sequence of operations to tag and isolate the necessary features. As a step in this direction, Bird [Bird84] has implemented a full compiler with grid data-types, formal operators, and control structures, in order to experiment with these image operators in our RPS environment. The compiler runs in the application layer of our system; its code generator is based on the simple stage-code generator implemented for the DRC experiments of the previous section.

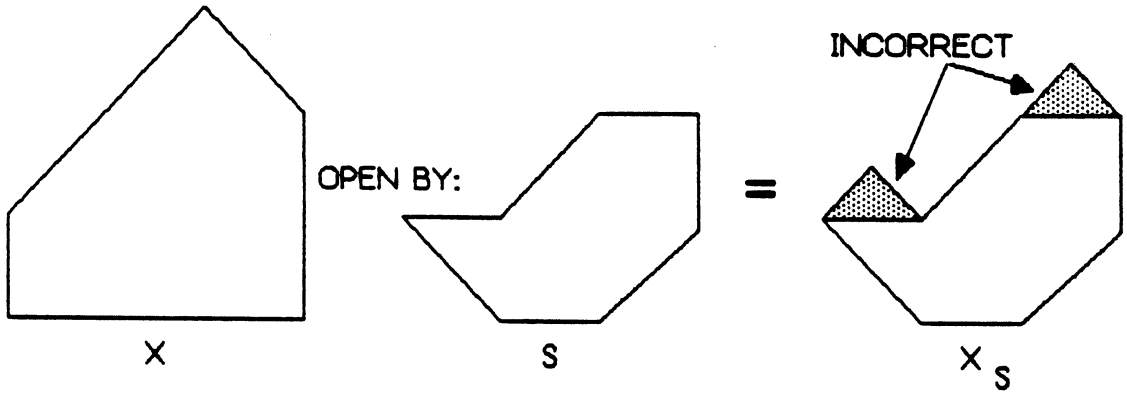


Figure 5.9 Attempt to Open X by S using Decoupled Computation

5.7. Summary

This chapter studies RPS-based DRC tools. Approaches to DRC systems are reviewed. It is suggested that a cellular DRC realized on RPS hardware is not a replacement for traditional DRC systems, but rather an alternative that sacrifices some quality for increased speed. We show that the formulation and operators of Sec. 3.2.2 provide a useful conceptual and notational tool for DRC algorithms. A simple width check is designed and implemented using this approach. Extensions to other mask operations are suggested.

CHAPTER VI

DESIGN TRADEOFFS FOR RPS-STRUCTURED DA ENGINES

6.1. Introduction

The previous two chapters explored routing and rules checking applications in an RPS environment. A principle component of the experimental work undertaken there was the identification of critical performance issues intrinsic to RPS systems. We have attempted to separate and abstract issues that appear to be generic to all RPS architectures from those endemic to our experimental hardware. With this background, this chapter proceeds to explore the architectural design of RPS-structured DA engines. Cost/performance tradeoffs and optimizations are analyzed for routing and design rule checking. As expected, we again partition the discussion into local design, in this case subarray stage functionality, and global design, particularly tradeoffs among different pipeline configurations.

6.2. Local Design of RPS-Structured DA Engines

This section focuses on the functionality of subarray stages appropriate to support routing and rules checking tasks. From our experimental systems we have identified several stage functions and subarray-computations needed to accommodate these tasks efficiently on an RPS machine. The functional inadequacies of a stage manifest themselves as compromises forced upon us as we attempt to map complex algorithms onto the stage hardware; the sub-optimal structure of the global router designed in Chapter IV is derived from such compromises. Given these experimental observations, we might begin immediately the design of an "optimal" stage architecture. In this section, we reject this approach and argue that to embark upon detailed

design at this point is necessarily naive for two reasons.

First, we do not yet have a sound, quantitative picture of the tradeoffs involved for RPS systems. Intuition and experimental data support these ideas: long pipelines are better than short ones; fast stages are better than slow ones; host overhead is better low than high. However, pipeline length, rate, and overhead are all parameters influenced by the stage design: overhead includes stage reprogramming (if indeed the stages are programmable); stage bandwidth depends on the complexity of stage functionality; pipeline length is influenced by the cost and size of individual stages. For example, the fastest stage may not be the one we can afford to use in the longest practical pipeline. Several designs may each suffice to meet a set of performance constraints, yet each design may incur a radically different cost.

Second, we have not yet specified the range of tasks this stage must handle. A stage for a routing engine is not necessarily optimal for a DRC engine. A stage to support both tasks likely represents a compromise. A general-purpose stage (roughly analogous to a bit-serial array node) is likely sub-optimal for each DA task, but perhaps cost-effective because of its wider range of applications.

To avoid these problems, we do not propose a single, detailed stage design for these DA tasks. Rather, we informally catalog those aspects of stage architecture desirable for routing and rules checking, as deduced from our experiments.

Fig. 6.1 shows again the basic structure of a subarray stage and its components: the buffering scheme, subarray storage, and subarray processor. Each is treated separately in the following sections.

6.2.1. Buffering Scheme

The buffers align the serial cell stream into subarrays for processing. A critical parameter is the datapath width: wider cells are needed to support DRC on several bit-planes, and multi-layer routing with complex path-functions. It is possible to allow tradeoffs between cell size and line-buffer length. Although Fig. 6.1 interprets the buffers as shift-registers ("delay" lines), in practice these are constructed of standard RAMs with access mechanisms to implement the

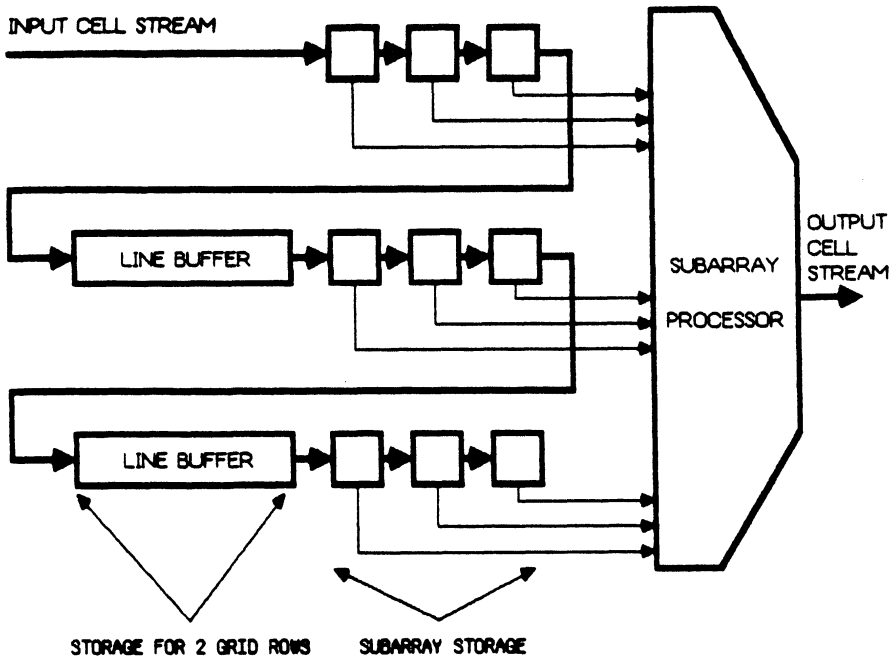


Figure 6.1 Basic Subarray Stage Structure

shift-by-one-cell function [Loug84]. With slightly more complex access mechanisms, we might consider the line buffers as a large collection of bits to be reformatted as necessary. This is a tradeoff reminiscent of virtual array architectures with large memories resident at each node. For example, assume 1 M-bit of line buffer memory in a 3×3 stage. This suffices to support grids with 65536 columns of 8-bit cells, 16384 columns of 32-bit cells, etc. It is expected that the overhead to deal with wider cells will degrade the bandwidth of the stage.

The next obvious constraint here is the external datapath width, i.e., the width the stage IO ports. Current cytocomputer stages have fixed 8-bit external and internal datapaths. A straightforward solution is time-multiplexed access to pieces of larger cells streamed sequentially through the narrower external datapath. Pieces of one cell accumulate in a shift register; when complete, the entire cell is moved into the line buffers.

The shift-register interpretation of Fig. 6.1 handles only rectangular grids. For routing, it is more desirable to process just the region of potentially expandable cells, which is never rectangular. Referring again to the perfect, unimpeded diamond-shape expansion wavefront in

Fig. 4.4, the capability to process only the area of expansion could reduce cell computations by up to 50%. Extraction of the appropriate polygon in the routing grid to be streamed through the stages is a standard raster-graphics operation. However, the stage must now know the shape of this non-rectangular region to handle its ragged edges: a one-cell shift no longer produces the next subarray at the edges of the region. More accesses to the subarray storage and buffers are necessary for proper alignment. Overall stage bandwidth may be degraded in order to handle the more complex alignment. This is a tradeoff that may be justified for a routing engine, but not for a DRC engine.

6.2.2. Subarray Storage

Subarray storage holds successive subarrays of data from the input grid, and represents the current "state" of the input for the subarray processor. Although large subarrays have some applications for DRC (see [Seil82]), the 3×3 structure is the most generally useful. The width of each cell in the subarray defines the widest cell that can be processed. For small subarrays, very wide cells are not expensive. The subarray must be accessible by the buffer mechanism, and by the subarray processor. For non-stateless applications which inject their results back into the input grid, it must be readable *and* writable by the subarray processor. The subarray should play a role analogous to the general registers in a general-purpose CPU: subarray cells are individually addressable, can function as source or destination for subarray-computations, and are implemented in a very fast technology. Unlike conventional CPUs, however, concurrent access to some or all cells is desirable for subarray-computations, especially with highly parallel subarray processors.

An extension to better support the DRC task is to permit individual bit-plane access to the subarray. That is, a $3 \times 3 \times 32$ bit subarray accessible as 9 32-bit cells, or 32 9-bit planes. Individual bit-planes (masks) are the appropriate primitive objects for which we should provide load/store access in a DRC.

6.2.3. Subarray Processor

The subarray processor produces the output cell stream based upon input data in the subarray storage, and upon any stored or computed data internal to the processor. After specifying the necessary subarray-computations (e.g., well-defined cell expansion or bit-plane DRC algorithms), cost and bandwidth constraints, several architectural tradeoffs can be made. We consider separately these four design issues: cell format, instruction set, parallelism in subarray computation, and global state.

- **Cell Format**

  As mentioned previously, wider cells are desirable to support more complex tasks. DRC demands several parallel bit-planes for masks and intermediate results. Routing demands flexible formats, essentially data structures with bits, integers, etc., coexisting within each cell. Cytocomputer stages are almost entirely table-driven: cell reformatting is accomplished by full-width 8-bit table lookup. Because this does not scale to larger cells, we must consider explicit sub-field access mechanisms. However, experience with the current hardware suggests that table lookup is an efficient, flexible mechanism for scalar calculations on small cells. Hence, we suggest a combination of the two: table lookup on the low-order bit positions of a cell, and a barrel shifter to move the desired bits into these low-order positions. Note the tradeoff here: larger tables process more bits in a cell, but require more time to load during stage reprogramming. 10-bit to 12-bit lookup seems practical.

- **Instruction Set**

  The basic tradeoff here is the choice between a single-purpose, hardwired stage and a programmable stage. Our experimental hardware has only a rudimentary instruction set consisting of the values of some tables and registers; dataflow is fixed and branch-free, and each computation is essentially atomic. Unless we must address precisely one, unique DRC or routing task, it appears that some form of instruction set is unavoidable. The size and complexity of the instruction set, and the speed at which individual instructions execute impact overall stage size and bandwidth. If the table-lookup mechanism just described is available to each instruction, the speed of the table memory constrains the

speed of each instruction. It appears practical to incorporate some flexibility in choosing the source and destination of operations on data in subarray storage, and some special-purpose instructions to address the needs of individual DA tasks, e.g., a cell-expand instruction for a particular routing formulation, or a bit-plane pattern-match operation similar to the mechanism in current cytocomputers.

- **Parallel Subarray Computation**

  Central to one subarray-computation is the process of transforming an input subarray (extracted from the input stream) into one cell value injected into the output stream. The issue to be addressed is the amount of parallelism in this computation. In cytocomputers, the basic mechanism of subarray-computation transforms a 3×3 subarray into a 9-bit table address; in our hardware the 9-bit vector is generated serially during minor-cycles of the stage clock, after which the table operations may be viewed as *parallel* computations on the subarray.

  There are speed/cost tradeoffs between a single, time-multiplexed datapath with appropriate storage for temporary results, and parallel hardware that performs the computation in one step. For DRC, a datapath similar to that of cytocomputers suffices. For routing with general path-functions, we require ALU-type functions and the previously mentioned sub-field access mechanism. Consider again the simple unit-cost router of Fig. 4.1 in which free cells are labeled with their distance from the source. Assume now that each cell stores a penalty value: expanded cells are labeled with the minimum *cost* of a path to the source. A cell with active neighbors requires the following sort of computation:

$$\begin{aligned} \text{new } \mathit{cost}(\text{cell}) := \text{minimum } \{ \; & \text{current } \mathit{cost}(\text{cell}), \\ & \mathit{cost}(\text{north}) + \mathit{penalty}(\text{cell}), \\ & \mathit{cost}(\text{south}) + \mathit{penalty}(\text{cell}), \\ & \mathit{cost}(\text{east}) + \mathit{penalty}(\text{cell}), \\ & \mathit{cost}(\text{west}) + \mathit{penalty}(\text{cell}) \; \} \end{aligned}$$

  Note the implementation alternatives: one general ALU plus temporary storage (serial processing), two ALUs plus storage (moderate parallelism), four ALUs plus minimum-detect hardware (fully parallel), etc. Again, the appropriate choice is dominated by cost and speed constraints.

- **Global State Information**

  For completeness, we mention again the need to retain internally global information computed as grid cells stream through the stage. We require counters to count specific events, index mechanisms to record the (x, y) grid address of specific events, and temporary storage to retain computed cell values. An attractive approach is to duplicate the subarray storage and its access mechanisms, and to allow any data in either subarray to participate as a source or destination for computations. This has the advantage of uniformity, allowing the same kinds of data (bit-planes, complete cells, etc.) to be stored in either subarray.

It is expected that the subarray processor dominates the size and complexity of the stage. Although treated separately here, the buffers, subarray storage, and processor must be tightly integrated. Conventional wisdom[1] suggests four implementation technologies, each embodying different tradeoffs: microprocessor-based, bit-slice, semi-custom or custom implementation. A microprocessor-based stage is flexible but slow. A bit-slice stage can be fast, but larger and more costly; see [RuMA84] for our early ideas about a bit-slice-style stage. A semi-custom stage can be fast, and less costly due to the integration of functions; moreover, it becomes less expensive to provide parallel hardware (e.g., parallel ALUs for a router) if the critical function to be replicated is implemented on a semi-custom chip. Custom implementation is faster than a microprocessor-based design given custom functionality, and the resulting small stage makes long pipelines practical. However, design cost is highest here. The following section quantifies some of the cost performance tradeoffs that must be known in order to make an appropriate choice among these implementation alternatives.

[1]This actually mirrors the hardware learning curve for machines in the cytocomputer family [Loug84]. Early subarray stages are mostly MSI technology, with some bit-slice components. Later stages use replicated semi-custom components for speed. Others are fully-custom for density and long pipelines.

6.3. Global Design of RPS-Structured Engines

This section explores the impact on performance of variation in three basic global parameters: pipeline length, stage rate, and pipeline overhead. Given the dependence of overhead upon specific hardware environments and interfaces, we continue the practice of previous chapters of inserting a single lumped overhead time τ_{ovh}, at the appropriate places in our models. Stage cycle time is τ_{stage}, pipeline length is S stages. Only 3×3 subarray stages are considered. For simplicity, it is assumed again that each subarray-computation (wavefront expand, DRC step) requires one atomic τ_{stage} cycle. The metrics of interest are execution time and cost/performance, appropriately defined for each task.

Our approach is to employ the execution time models developed previously to study the effects of variation in global parameters. Wherever possible we strive to obtain analytical expressions for performance. By varying parameters and solving these models, we obtain hardware configurations embodying different cost and performance tradeoffs. If inflexible constraints on size and speed are imposed, we may select just those configurations that satisfy them. These differing configurations are then candidates for physical implementation.

We shall sometimes study optimization issues with respect to precisely one, single task, e.g., with respect to the gate array benchmark of Chapter IV. It is not suggested that the configuration optimal for such a narrow problem is the right one for general use. Rather, it is suggested that this is an appropriate technique to gain insight into the behavior of RPS systems. We expect that the designer of an RPS-structured engine has some *a priori* knowledge of the structure of the class of DA problems to be attacked. Conducting analyses on a *family* of representative tasks yields data about specific design compromises that best accommodate the entire range of expected tasks. In the following, we treat routing tasks first, and then DRC.

A starting point for the analysis of maze-routers is the analysis of single wires. Throughout this discussion, it is assumed that routing time is dominated by expansion time, i.e., backtrace, cleanup are negligible. Hence, we shall refer to expansion time as simply "routing time". The central problem here is optimal framing, treated thoroughly in Chapter IV. In particular, we have already studied the effects of variation in relative overhead τ_{ovh}/τ_{stage} in

Sec. 4.7.2.2 using the dynamic programming solution to the framing problem. To complete this analysis, we now consider the effects of variations in τ_{stage}.

Consider an experiment similar to the one in Sec. 4.7.2.2 that studied sensitivity with respect to overhead. Our goal is to study tradeoffs between pipeline length and speed. We solve the optimal framing problem for the following task and hardware configurations:

L	=	32, 256 cells (orthogonal 2-point nets from Fig. 4.13)
S	=	2, 8, 32, 64 stages
τ_{ovh}	=	10μs
τ_{stage}	=	10ns, 50ns, 100ns, 500μs, 1μs, 5μs, 10μs, 50μs

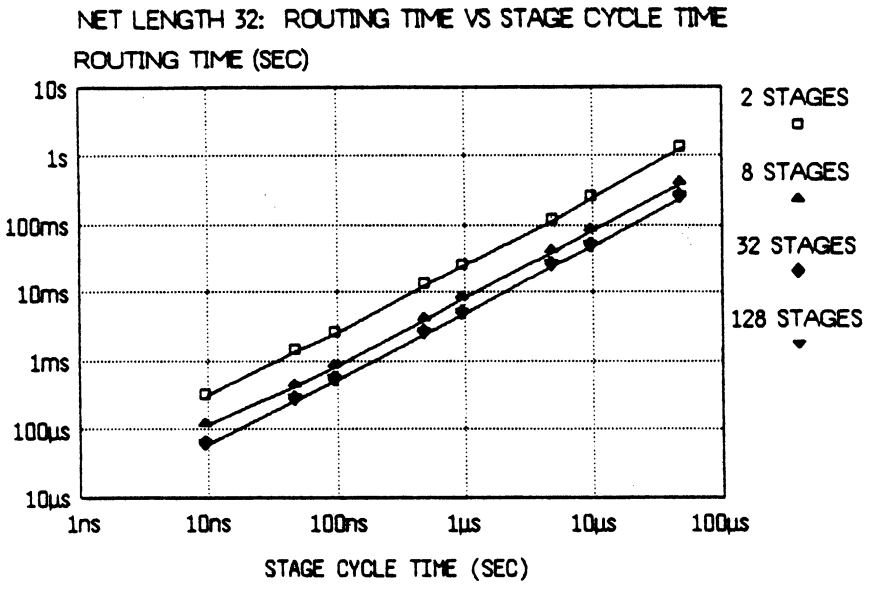
This experiment varies wire length, pipeline length, and stage rate over wide regions of feasible values, and computes the optimal routing time for each set of parameters. The results are shown in Fig. 6.2. For each wire length, we plot total time versus τ_{stage}, parameterized by pipeline length.

As expected, times improve with decreasing τ_{stage}, increasing S. Long pipelines are useful for long wires, but not for short ones: the curves for $S=32$, $S=28$ are identical for a wire length of 32 cells. The curves shown are very close to straight lines (log log scale), confirming that for a fixed net geometry, time varies inversely with stages, directly with τ_{stage}. (Recall from the sensitivity studies of Fig. 4.29, however, that the optimal framing strategy that yields the optimal time is likely to change *significantly* as τ_{stage} changes.) Hence, we can make some of the intuitive tradeoffs between stage rate and pipeline length.

The next issue to address is the choice of optimal pipeline length for an entire routing task. Suppose that the routing time for a net of length L is just L^3/S, and that a routing task consists of N_w wires of lengths $\{L_i\}_{i=1}^{N_w}$. Then the total routing time for the task is

$$T_{RPS} = \sum_{i=1}^{N_w} \frac{L_i^3}{S} = \frac{\text{constant}}{S} \tag{6.1}$$

which varies inversely with S. A decrease in routing time with additional stages is expected, but given our intuition about diminishing returns from stages, we might expect there to be some *best* pipeline length. Clearly, there is an upper bound on useful pipeline length at $S = \max \{L_i\}_{i=1}^{N}$. No pipeline with more stages than the length of the longest wire is ever useful. An obvious guess is to take $S = \bar{L}$, the mean wire length. But it is generally true that



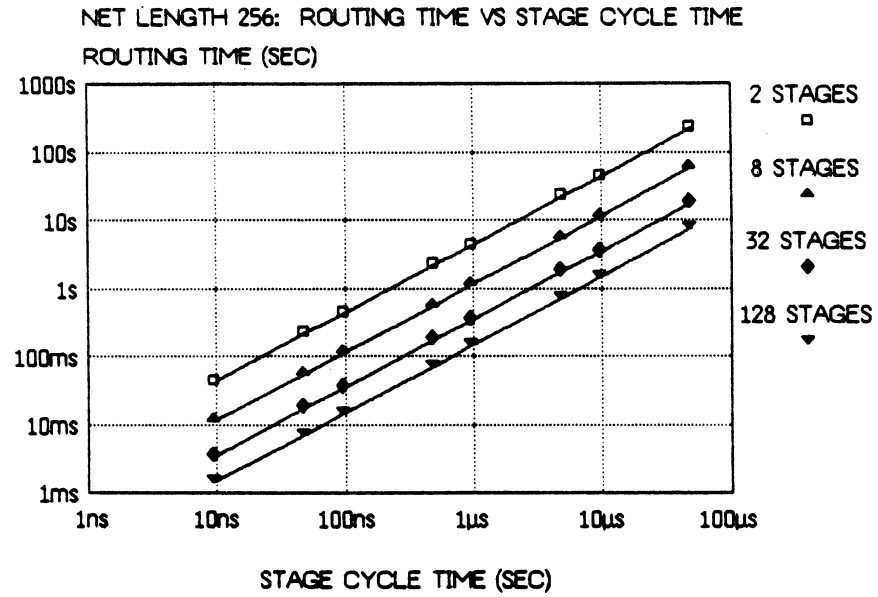


Figure 6.2 Routing Time Variation with Stage Cycle Time

The optimum pipeline length S_{opt} minimizes this quantity.

We consider the task of gate array routing, which appears to be a viable application for RPS systems from our experiments of Sec. 4.5.3. This choice is motivated by two facts. First, maze-routers have been routinely employed for gate arrays, e.g., [High83]. Second, the distribution of wire-lengths has been characterized theoretically for gate array routing. Assuming that a Rent's Rule relationship[2] holds for the placement of logic in a square gate-array, [Dona81] derives theoretically and verifies experimentally that the fraction $f(k)$ of wires with length k in a square gate array is distributed as

$$\begin{aligned} f(k) &= gk^{-\gamma} && 1 \le k \le L_{max} \\ &\approx 0 && k > L_{max} \end{aligned} \tag{6.3}$$

where g, γ, and L_{max} are constants. γ is in the range $1 \le \gamma < 3$.

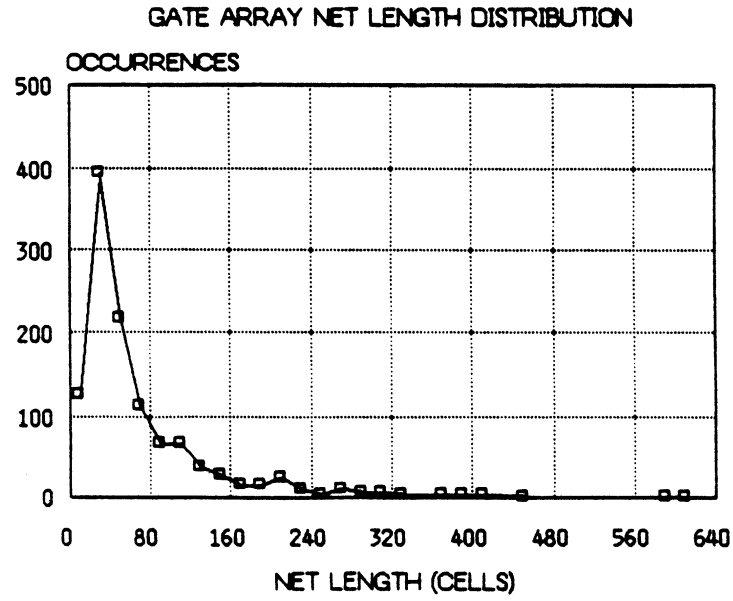
Fig. 6.4 shows the measured distribution of wire lengths for the 900 gate array routed in Sec. 4.5.3. Also shown is a least-squares fit to the distribution of (6.3), yielding parameters $g \approx 1861$, $\gamma \approx 1.883$. Note the good fit except for very short wires, which results from the fact that our gate array is not actually square as assumed by the theory (it is 18 $\times$ 25 blocks).

With a cost/performance metric and characterization of the routing task, we can formulate precisely the question of optimal pipeline length. We need these assumptions:

- A routing task has N_w wires of lengths $\{L_i\}_{i=1}^{N_w}$ distributed as in (6.3). In particular, the number of wires with length L is $N_w f(L)$.
- The time $T_{RPS}(L, S)$ to route a wire of length $L \le S$ is $L^2 \tau_{stage}$. This ignores latency and overhead, and assumes a rather rudimentary framing: the wire lies in an $L \times L$ square.
- The time $T_{RPS}(L, S)$ to route a wire of length $L > S$ is $\frac{L^3}{S} \tau_{stage}$, i.e., $\frac{L}{S}$ passes of L^2 cells through the pipe. Again, no latency or overhead, and only rudimentary framing.

---

[2]Rent's Rule [LaRu71] relates the number of circuits in a logic graph to the number of IO terminals. It states that the average number of terminals T per group of C circuits is: $T = AC^p$. A and p are constants, and p is called the Rent exponent. [Dona81] shows that in the distribution of wire lengths, $\gamma \approx 3 - 2p$. For p in the expected range $1/2 \le p < 1$, we actually have $1 < \gamma \le 2$.



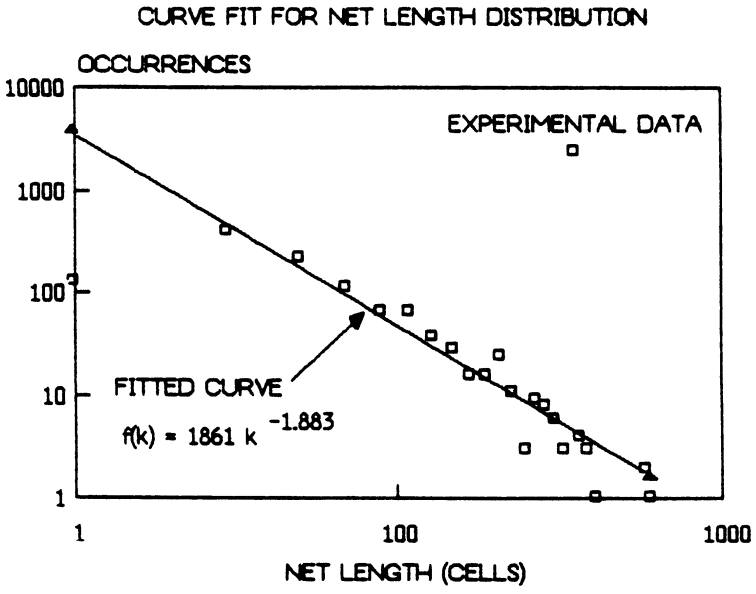


Figure 6.4 Net Length Distribution and Curve Fit for Gate Array Benchmark

The total time T_{RPS} to route this array is then:

$$T_{RPS} = \sum_{i=1}^{N_w} T_{RPS}(L_i, S) \tag{6.4}$$

However, it is more useful to sum not over individual wires, but over individual net lengths:

$$T_{RPS} = \sum_{k=1}^{L_{max}} N_w f(k) T_{RPS}(k, S) \tag{6.5}$$

Recalling that the time to route length $L \leq S$ is different from the time for $L > S$, we partition the sum and substitute for $T_{RPS}(k, S)$, and $f(k)$ accordingly

$$\begin{aligned} T_{RPS} &= \sum_{k=1}^{S} N_w f(k) k^2 \tau_{stage} + \sum_{k=S+1}^{L_{max}} N_w f(k) \frac{k^3}{S} \tau_{stage} \\ &= \sum_{k=1}^{S} N_w g k^{-\gamma} k^2 \tau_{stage} + \sum_{k=S+1}^{L_{max}} N_w g k^{-\gamma} \frac{k^3}{S} \tau_{stage} \end{aligned} \tag{6.6}$$

which simplifies to become:

$$T_{RPS} = N_w \tau_{stage} g \left[\sum_{k=1}^{S} k^{2-\gamma} + \frac{1}{S} \sum_{k=S+1}^{L_{max}} k^{3-\gamma} \right] \tag{6.7}$$

Since γ is a real number, we approximate these sums by their integrals:

$$\begin{aligned} \sum_{k=1}^{S} k^{2-\gamma} &\approx \int_{1}^{S+1} k^{2-\gamma} dk = \frac{(S+1)^{3-\gamma} - 1}{3-\gamma} \approx \frac{S^{3-\gamma} - 1}{3-\gamma} \\ \sum_{k=S+1}^{L_{max}} k^{3-\gamma} &\approx \int_{S+1}^{L_{max}+1} k^{3-\gamma} dk = \frac{(L_{max}+1)^{4-\gamma} - (S+1)^{4-\gamma}}{4-\gamma} \approx \frac{L_{max}^{4-\gamma} - S^{4-\gamma}}{4-\gamma} \end{aligned} \tag{6.8}$$

whereupon T_{RPS} simplifies to become:

$$T_{RPS} = N_w \tau_{stage} g \left[\frac{S^{3-\gamma}}{(3-\gamma)(4-\gamma)} + \frac{L_{max}^{4-\gamma} S^{-1}}{4-\gamma} - \frac{1}{3-\gamma} \right] \tag{6.9}$$

Substituting (6.9) into the expression for cost/performance (6.2), we obtain finally the desired result:

$$\frac{\text{cost}}{\text{performance}} = cp = (\alpha S + \beta) \tau_{stage} g \left[\frac{S^{3-\gamma}}{(3-\gamma)(4-\gamma)} + \frac{L_{max}^{4-\gamma} S^{-1}}{4-\gamma} - \frac{1}{3-\gamma} \right] \tag{6.10}$$

where we use cp to denote cost/performance.

A pipeline is optimal if (6.10) is minimum. To search for minima in the range of feasible S, $[1, L_{min}]$, assume S is continuous. The derivative is then:

$$\frac{d(cp)}{dS} = \tau_{stage}\, g \left[\frac{\alpha}{3-\gamma} \left(s^{3-\gamma} - 1 \right) + \frac{\beta}{4-\gamma} \left(S^{2-\gamma} - L_{max}^{4-\gamma} S^{-2} \right) \right] \tag{6.11}$$

Note that the derivative changes sign across the interval $[1, L_{max}]$:

$$\begin{aligned} \left. \frac{d(cp)}{dS} \right]_{S=1} &= \frac{\tau_{stage}\, g \beta}{4-\gamma} \left(1 - L_{max}^{4-\gamma} \right) < 0 \\ \left. \frac{d(cp)}{dS} \right]_{S=L_{max}} &= \frac{\tau_{stage}\, g \alpha}{3-\gamma} \left(L_{max}^{3-\gamma} - 1 \right) > 0 \end{aligned} \tag{6.12}$$

Next, note that the second derivative is strictly positive in the interval

$$\frac{d^2(cp)}{dS^2} = \tau_{stage}\, g \left[\alpha S^{2-\gamma} + \frac{2-\gamma}{4-\gamma} \beta S^{1-\gamma} + \frac{2\beta}{4-\gamma} L_{max}^{4-\gamma} S^{-3} \right] > 0 \tag{6.13}$$

as this has only positive terms for $\gamma < 2$, which we assume to be commonly true. $\frac{d(cp)}{dS}$ is thus monotone increasing. This monotonicity and the sign change prove that there is a unique cost/performance minimum (a zero of the derivative) for S in $[1, L_{max}]$. Hence, there is indeed an optimum S.

Unfortunately, these equations are sufficiently complex to preclude a simple closed form for S_{opt}. However, it is straightforward to verify empirically the following approximation:

$$\begin{aligned} S_{opt} &\approx \left[\frac{\beta(3-\gamma)}{\alpha(4-\gamma)} \right]^{\frac{1}{\delta-\gamma}} L_{max}^{\frac{4-\gamma}{\delta-\gamma}} \qquad \text{for } \frac{\beta}{\alpha} \text{ small} \\ &\approx L_{max} \qquad\qquad\qquad\qquad\quad \text{for } \frac{\beta}{\alpha} \text{ large} \end{aligned} \tag{6.14}$$

Thus, S_{opt} increases as a small power of β/α. Using the parameters from our gate array (Fig. 6.4), we illustrate the expected shape of these cost/performance curves in Fig. 6.5. S_{opt} increases with increasing β/α, which has a reasonable physical interpretation. α is the cost of one stage, and β the cost of all other hardware overhead. As α decreases relative to β (increasing β/α), the cost of each stage decreases measured against this fixed overhead. Hence, as stages become less costly, longer pipelines become more cost-effective: S_{opt} increases toward L_{max}. Note that for large, complex stages, β/α is small; for highly integrated stages β/α is

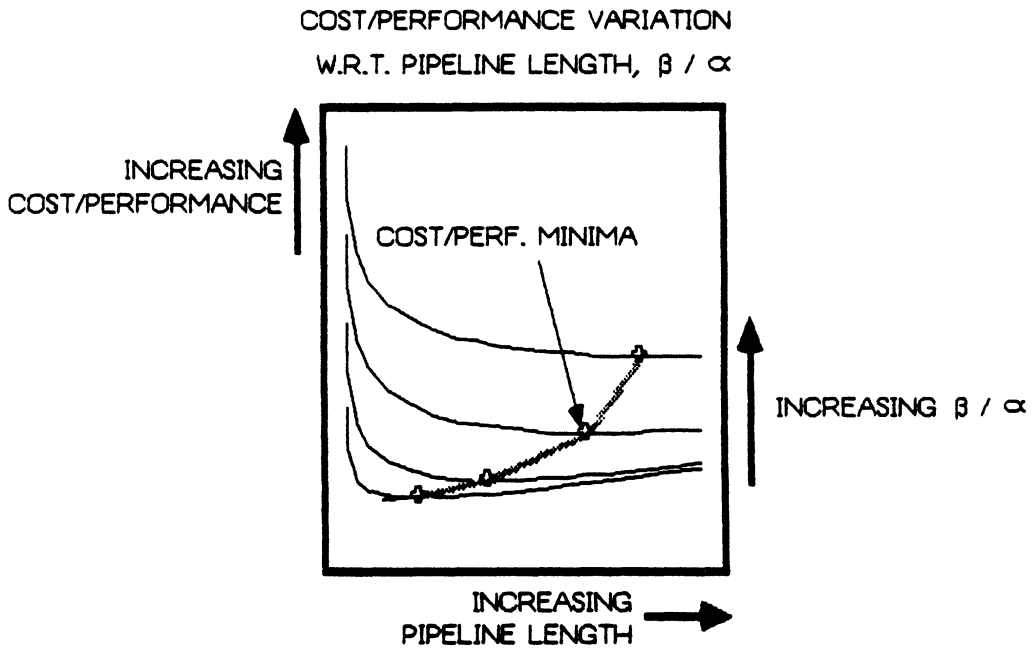


Figure 6.5 Predicted Shape of Routing Cost/Performance Curves

large.

To verify that the primitive assumptions used in this analysis do not destroy the validity of the result, we consider again the gate array benchmark of Chapter IV. Fig. 6.6 shows the cumulative distribution function for net lengths in the gate array, and a quantized approximation to this distribution. For each of the small number of unique net geometries in the quantized version we solve for optimal routing time as a function of pipeline length. The dynamic programming solution for optimal framing of Chapter IV is employed to compute these times. From these results, we estimate the complete routing time for a given pipeline length as a weighted sum of the individual net routing times. With complete routing times known for various pipeline lengths, we construct cost/performance curves. For this experiment, we examine the following hardware configurations:

$$
\begin{aligned}
S &= 1, 4, 16 \text{ stages, and } 32k \text{ stages, } k = 1, 2, \ldots, 16 \\
\tau_{stage} &= 1\ \mu s \\
\tau_{ovh} &= 1 \text{ ms}
\end{aligned}
$$

where the large number of S values is chosen to insure sufficient resolution to identify

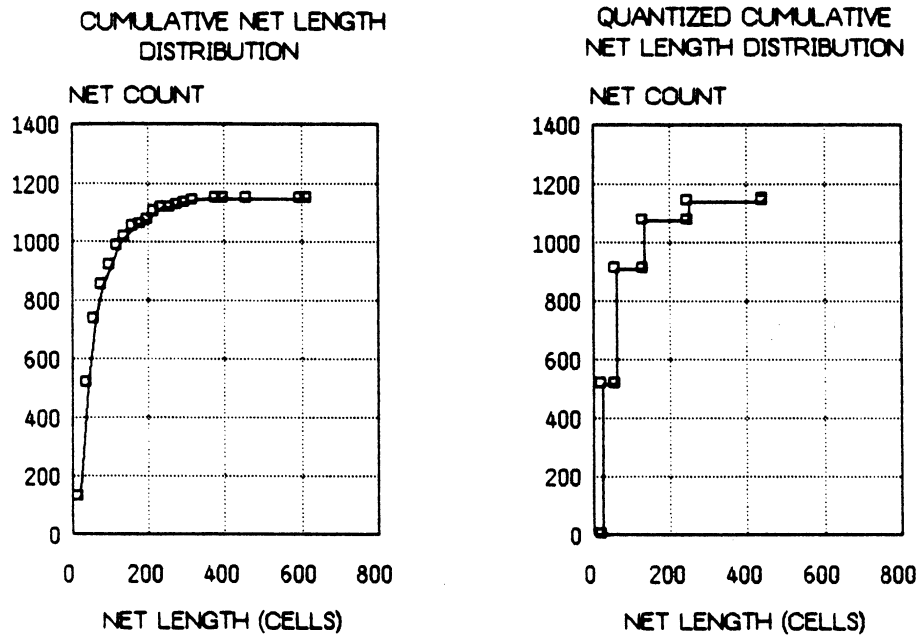
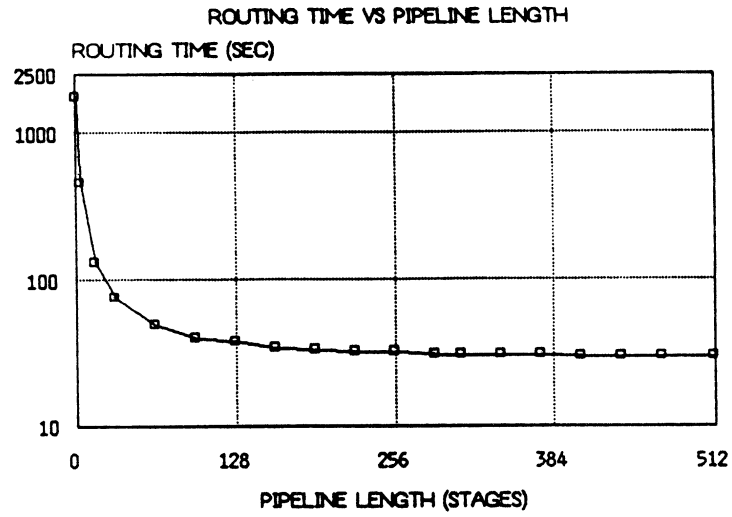


Figure 6.6 Measured and Quantized Cumulative Length Distributions for Gate Array Benchmark

cost/performance minima.

Fig. 6.7 shows both total routing time and cost/performance variation with pipeline length. As expected, routing time decreases with additional stages, reaching an asymptote as pipeline length increases past the maximum net length. Note how the rate of improvement decreases with large S, confirming our intuition about diminishing returns. The cost/performance curves for various β/α match nicely the shapes predicted by analysis in Fig. 6.5. Minima are pronounced for small β/α, and are almost indiscernible for large β/α as the curves become flat for large S. As a practical matter then, for β/α large, we do not need to attain the precise minimum at S_{opt} to obtain good performance; a point slightly past the knee of the curve where the cost/performance improvement is deemed "flat enough" should suffice.

The design rule checking task admits a similar analysis. It is, however, simpler in the sense that there are no framing problems, the required processing steps are fewer (some portion of a routing grid must visit a stage for each cell in a routed wire; we expect a DRC to have fewer processing steps), and processing is homogeneous across the grid.



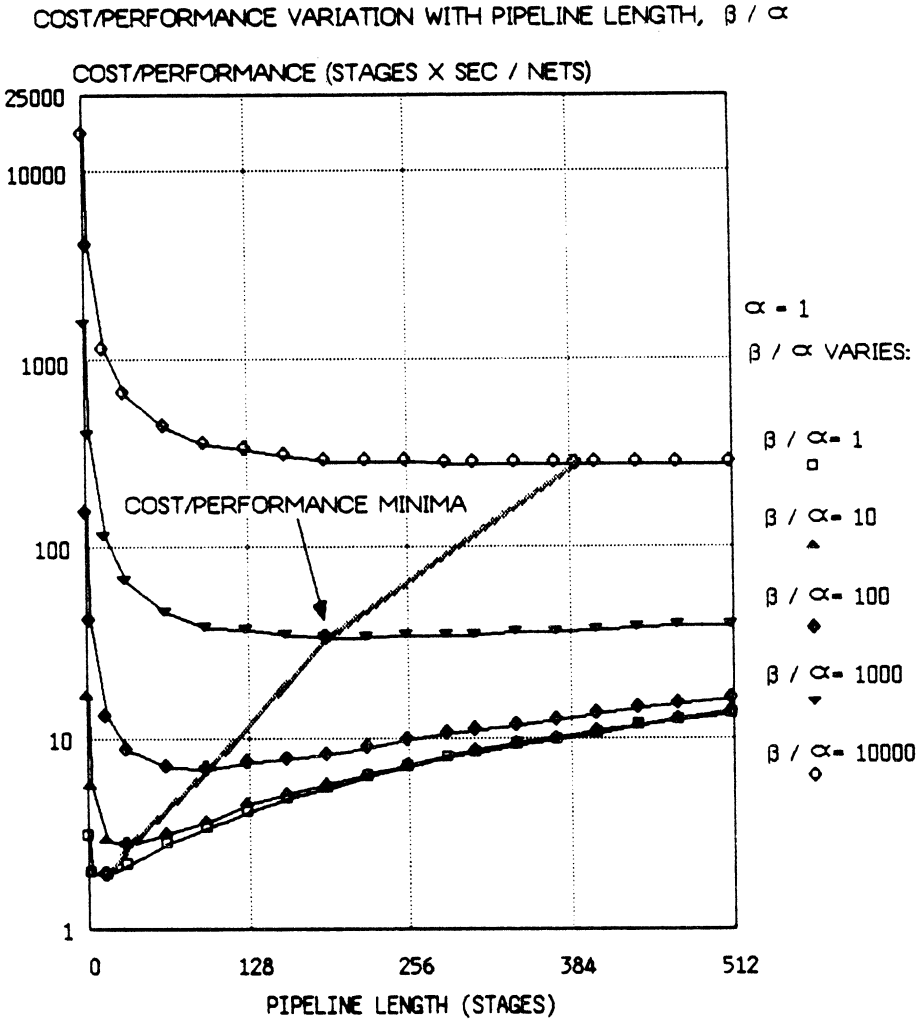


Figure 6.7 Gate Array Routing Time, Cost/Performance Variation

Suppose a DRC requires K subarray-computations. Assume as usual a 3×3 subarray stage, and for simplicity, assume that we only process square grids. The time $\tau_{pass}(r, S, X)$ to process an $X \times X$ grid r passes through an S-stage pipeline is reasonably modeled as

$$\tau_{pass}(r, S, X) = r\left[\tau_{ovh} + (S(X+2) + X^2)\tau_{stage}\right] \tag{6.15}$$

where we assume that the overhead τ_{ovh} (potentially large for a large mask) is incurred on *each* pipeline pass.

It is straightforward to show that the minimum time to process this DRC occurs with a K-stage pipeline. For simplicity, assume that if $S < K$, then K is a multiple of S, i.e., exactly K/S passes are required for the DRC. If the grid size is $X \times X$ cells, the ratio of DRC processing times for a K-stage pipe and a shorter pipe is

$$\begin{aligned} \frac{T_{DRC}(\ K \text{ stages})}{T_{DRC}(\ S<K \text{ stages})} &= \frac{\tau_{pass}(1, K, X)}{\tau_{pass}(\frac{K}{S}, S, X)} \\ &= \frac{\tau_{ovh} + (K(X+2) + X^2)\tau_{stage}}{\frac{K}{S}\tau_{ovh} + \frac{K}{S}(S(X+2) + X^2)\tau_{stage}} < 1 \end{aligned} \tag{6.16}$$

which demonstrates that K stages has the best attainable time.

Cost/performance can be defined as before. We take "DRC tasks per unit time" to be the performance metric, i.e., T_{DRC}^{-1}. Assuming cost is again $\alpha S + \beta$ we find:

$$\frac{\text{cost}}{\text{performance}} = cp = (\alpha S + \beta)\left\lceil\frac{K}{S}\right\rceil(\tau_{ovh} + [S(X+2) + X^2]\tau_{stage}) \tag{6.17}$$

It is not always the case that there is a minimum cp for $S<K$. For some combinations of parameters, cost/performance decreases monotonically throughout the interval $[1, K]$; as with simple execution time, K stages has the best attainable performance. However, examination of the derivative shows that if the following condition is satisfied:

$$\eta < \frac{\beta}{\alpha} < K^2\eta \quad \text{where } \eta = \frac{X\tau_{stage}}{\tau_{ovh} + X^2\tau_{stage}} \tag{6.18}$$

then, just as with analysis for gate array routing, there is a unique S_{opt} *within* $[1, K]$ that optimizes cost/performance. In this case, the DRC cost/performance curves again resemble those in Fig. 6.5.

Finally, note that throughout this section we have assumed the existence of only a single pipeline. An architecture with multiple, parallel pipelines is attractive for tasks with separable sub-tasks. In particular, a DRC is typically a concatenation of separate mask checks, each requiring only a short pipeline. Short, non-interacting (non-overlapping) nets in a routing task might likewise be routed in parallel, similar to the manner in which [NHLV82] suggests that non-interacting nets can be routed concurrently on an array machine. Fig 6.8 suggests a simple parallel pipeline arrangement. Multiplexors at the end of each pipe allow short pipes to be concatenated to form longer pipes as needed, e.g., to route long wires. This architecture requires only the addition of the multiplexors and a format-processor which takes each pipeline's output stream and reformats it into one single cell stream[3]. Conceivably, multiple designers can share the pipelines in such a machine if the necessary multiple IO channels are provided.

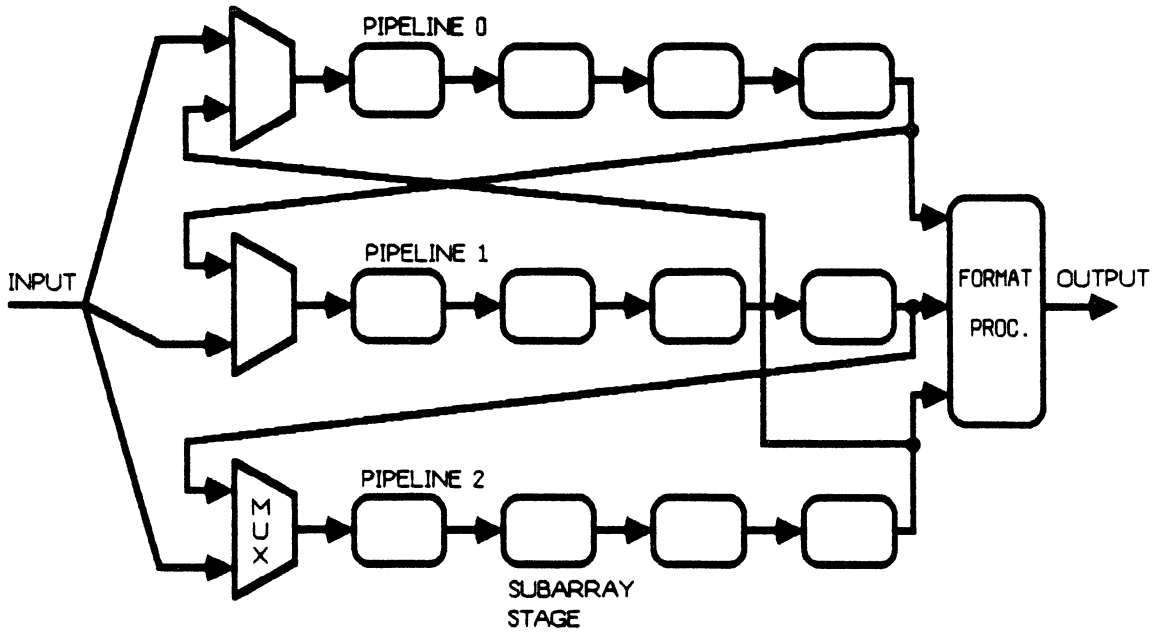


Figure 6.8 Parallel Pipeline Architecture

[3]Later machines in the cytocomputer family, while not yet employing parallel pipelines, do have one additional "combiner" processor for simple dyadic operations on two input cell streams to produce one output stream [Loug84].

We suspect that the previous cost/performance analyses can be extended to the multiple-pipeline case. We must characterize how the entire DRC is partitioned into individual mask checks, and can perhaps employ Rent's Rule to determine how non-overlapping nets are distributed across the routing grid.

6.4. Summary

This chapter studies the architectural design of RPS-structured DA engines, and focuses on routing and DRC tasks. Based on the experimental work of Chapters IV and V, we catalog informally several aspects of subarray-stage functionality appropriate to support these DA tasks. Appropriate cost/performance metrics are then formulated and treated analytically. Results from these studies are intended to be illustrative of those necessary to inform the designer of a practical RPS-structured DA machine. The models and techniques presented allow us to generate candidate hardware configurations (by varying model parameters) to meet practical cost or speed constraints.

CHAPTER VII

CONCLUSIONS

7.1. Summary and Contributions

The increasing complexity of designing VLSI systems makes increasingly attractive the design of special-purpose computer architectures to attack DA tasks. This thesis proposes the Raster Pipeline Subarray class as a model for DA engines. DA tasks that are well-represented on a cellular grid and characterized by strongly local functional dependencies among cells are candidate applications for RPS-structured DA machines.

The RPS organization is a systolic structure originated in picture-processing applications. Our formulation of the RPS class is the necessary abstraction from which to proceed toward more general problem domains such as DA. To study the viability of RPS-structured DA engines, we have constructed a complete, experimental RPS hardware/software environment around existing RPS hardware. This environment allows us to design and debug experimental RPS-based DA tools. A principal component of our experimental work is the identification of critical performance issues generic to all RPS systems.

The major application treated in this thesis is maze-routing. Complete, functional maze-routers have been implemented and optimized in our RPS environment. The successful routing of several large benchmarks proves the feasibility and practicality of RPS routing engines. Some of our systems have already exhibited performance superior to their software counterparts. This investment in experimental work has allowed us to identify and model several performance issues critical to real RPS systems.

The other application studied is design rule checking. We have shown that this task is particularly well suited to manipulation by mathematical image operators. Again, experimental benchmarks indicate the feasibility of RPS-based DRC engines.

The significant advantage of the RPS organization is its modularity. Hence, existing RPS engines, including our prototypes, can be improved incrementally and gracefully: additional pipeline stages improve execution time; additional cell memory enables larger problems to be tackled directly. Experiments over a range of pipeline lengths, and extrapolations based on experimental measurements support this claim.

We also study the design of RPS machines for specific DA problems. Based on our experimental systems, we have cataloged informally several aspects of subarray-stage functionality appropriate to support routing and DRC tasks. Performance models developed to analyze the behavior of our experimental systems are useful here as design tools for RPS architectures. Using these models, we study the sensitivity of performance metrics to changes in the hardware environment. By varying model parameters, we generate configurations with widely different cost and performance characteristics. Configurations that meet the necessary constraints are candidates for implementation. When appropriate performance metrics can be defined and problems characterized mathematically, we treat the optimization of pipeline length analytically.

The major contributions of this thesis include:

- A comparative survey of cellular architectures and DA machines showing that many engines designed to address grid-based DA problems can be regarded as instances of more general cellular organizations.
- Abstraction of the RPS class based on the central idea of pipelined serial subarray computation introduced by machines such as the cytocomputers. This abstraction, freed of application-specific implementation details, is the right level from which to proceed toward more general problem domains such as DA.
- Comparative performance analysis of RPS-structured and array-structured machines showing regions of shared performance. In particular, we have analyzed how basic features of

the RPS organization address central design issues for practical special-purpose DA hardware, and we have shown that it is not the case that these desirable engineering properties necessarily produce a comparative performance degradation of several orders of magnitude.

- The design and implementation of a layered software environment built around existing RPS hardware. This experimental RPS environment provides the tools necessary to build and study real RPS-based DA tools.
- Successful design and implementation of RPS-based maze-routers. PCBs and gate arrays have been successfully routed.
- Identification and rigorous analysis of the framing problem for RPS maze-routers. An approximate algebraic solution, and an optimal dynamic programming solution have been derived; these compare favorably with the experimental measurements of the constant-increment framing heuristic.
- Characterization of the sensitivity of single-net, fixed-geometry routing time to variations in pipeline length, stage rate, and pipeline overhead.
- Optimization of cost/performance metrics with respect to pipeline length for mathematical characterizations of gate array routing and DRC tasks.
- A catalog of several subarray stage functions and subarray-computations necessary to support efficiently routing tasks or DRC tasks on an RPS-structured engine. In particular, the stateless stages useful for picture-processing tasks have been shown to be an inappropriate, overly restrictive model on which to base a DA stage architecture.

7.2. Extensions and Future Research

We have already discussed several extensions to our work. These extensions have been distributed throughout this dissertation and appear in the individual sections to which they are relevant. These include: tuning and optimization of the current RPS routers; enhancements to the dynamic programming model for optimal framing; extensions to support 45° mask geometry; implementation of geometric query processing for IC masks based upon formal

image operators; alternative stage architectures to support routing and DRC; multiple-pipeline machines, and their related optimization issues. Two areas stand out as fruitful candidates for future research: the design and implementation of a DA stage, and the exploration of new DA tasks for RPS machines.

The experiments, tools, and techniques developed in this thesis establish a sound, quantitative basis from which to begin design of a practical DA stage. Realistic tradeoffs among cost, speed, maintainability, upgradability, range of application, etc, can be evaluated by employing or extending our models. In particular, the success of our maze-routers running in a short-pipe, high-overhead environment argues persuasively for the design of a production-quality routing engine.

Exploration of other promising DA problems is another important area of research. Physical DA is an especially rich area for study. We are particularly intrigued by the idea of parallel placement [UeKH83, ChBr83] and its realization on an RPS architecture. Clearly, the concurrent local module interchanges can be performed as subarray-computations. We assume that the grid stores tokens representing individual modules, and that the subarray processor stores the necessary connectivity information and absolute location of other modules. The serious problem is how to propagate the necessary information about interchanges to other subarray-processors, allowing us to employ a pipeline of interchange processors. We are uncertain if there is a practical resolution to this problem.

We believe it to be a fallacy that special hardware is only suitable for replacing entire processing steps (placement, routing, DRC, etc) in the descent-through-levels model of design. Rather, there are useful functions to be performed by more anonymous machines whose task is to accelerate, not to replace, portions of critical tasks. For example, Heller has suggested to us that instead of employing an RPS router to synthesize (route) a layout, we might employ it as an analysis tool to evaluate placement [Hell83]. Rather than expend effort optimizing an RPS design to maximize net completions, we might settle for a "good enough" router. This machine produces now a vector whose components are the wire lengths of each routable wire. The aggregate extent to which wires deviate from their minimum lengths is known from experimental studies of wireability to be a measure of the quality of a placement. Indeed, part of a

standard placement process is a crude routing phase composed of estimates of expected wire length and channel congestion. The addition of a very rapid, less crude routing phase as as part of the placement phase could improve placement quality and reduce time spent attempting to route an inherently bad layout. We suspect that there are other DA applications in both synthesis and analysis for which an RPS-structured machine is appropriate.

APPENDICES

APPENDIX A

LOCAL ALGORITHMS FOR TWO-LAYER ROUTER

This appendix presents the details of the wavefront expansion and cleanup algorithms that perform two-layer maze-routing in an RPS pipeline. The backtrace algorithm is omitted, as it is more similar to the one-layer backtrace, and is also performed on the host.

Let h(cell), v(cell), via(cell) be the horizontal, vertical, and via symbols respectively in the 3-tuple representation of a grid cell. The two-layer wavefront expand algorithm, partitioned into three subarray-computations, is as follows.

Given: A 3×3 subarray of cells in a routing grid.
Task: Expand subarray center on layer-1, layer-2, possibly introducing a via.
Algorithm: Wavefront_Expand_2_Layers
begin

Subarray_Computation_1:

```
/* escape on layer-1 from source or via, or extend
 * horizontal layer-1 segment starting at source or via
 */
if h(center) = h-free
then begin
        if h(north) in { h-src, h-via, ←h0, →h0 }
        then h(center) := ↑h1
        else if h(south) in { h-src, h-via, ←h0, →h0 }
        then h(center) := ↓h1
        else if h(west) in { h-src, h-via, ←h0, →h0 }
        then h(center) := ←h0
        else if h(east) in { h-src, h-via, ←h0, →h0 }
        then h(center) := →h0
```

Subarray_Computation_2:

```
/* layer-1 jogs not at source or via */
```

```
                else if h(north) = ↑h1
                then h(center) := ↑h2
                else if h(south) = ↓h1
                then h(center) := ↓h2
                else if h(west) in { ↑h1, ↓h1, ↑h2, ↓h2, ←h1 }
                then h(center) := ←h1
                else if h(east) in { ↑h1, ↓h1, ↑h2, ↓h2, →h1 }
                then h(center) := →h0
        end

Subarray_Computation_3:

        /* escape on layer-2 from source or via, or extend
         * vertical layer-2 segment starting at source or via
         */
        if v(center) = v-free
        then begin
                if v(north) in { v-src, v-via, ↑v, ↓v, ←v, →v }
                then v(center) := ↑v
                else if v(south) in { v-src, v-via, ↑v, ↓v, ←v, →v }
                then v(center) := ↓v
                else if v(west) in { v-src, v-via }
                then v(center) := ←v
                else if v(east) in { v-src, v-via }
                then v(center) := →v
        end

        /* generate via, expand onto alternate layer */

        /* from layer-1 to layer-2 */
        if ( via(center) = via ) and ( v(center) = v-free )
                and ( h(center) in { ←h0, →h0, ↑h1, ↓h1, ←h1, →h1, ↑h2, ↓h2 } )
        then v(center) := v-src

        /* from layer-2 to layer-1 */
        if ( via(center) = via ) and ( h(center) = h-free )
                and ( v(center) in { ←v, →v, ↑v, ↓v } )
        then h(center) := h-src

end
```

Backtrace is again done on the host, and is similar to the one-layer version except that a trace can cross layers at a via. Wire segments are relabeled as *blocked* on the appropriate layer, and vias are labeled with the 3-tuple (*via, h-via, v-via*).

Cleanup performs the standard removal of extraneous arrow labels, and also enforces via-exclusion. Cells adjacent in the four compass directions to a newly traced via are marked *novia* so that future vias are not embedded too close. This is a common physical restriction for vias

in PCB or VLSI technologies. This algorithm requires two subarray-computations.

Given: A 3×3 subarray of cells in a routing grid.

Task: Remove extraneous expansion labels, mark path as obstacle, enforce via-exclusion around subarray center.

Algorithm: Cleanup_2_Layer

```
begin

Subarray_Computation_1:

        /* remove extraneous labels, mark vias as obstacles */
        if h(center) in { ←h0, →h0, ↑h1, ↓h1, ←h1, →h1, ↑h2, ↓h2 }
        then h(center) := h-free
        if v(center) in { ←v, →v, ↑v, ↓v }
        then v(center) := v-free
        if ( h(center) = h-via ) or ( v(center) = v-via )
        then begin
                then h(center) := h-blocked
                then v(center) := v-blocked
                then via(center) := via
        end

Subarray_Computation_2:

        /* test for via in center or adjacent cell */
        center_via := false
        adjacent_via := false
        for cell := center, north, south, east, west do
        begin
                if ( h(cell) = h-blocked ) and ( v(cell) = v-blocked )
                and ( via(cell) = via )
                then begin
                        if cell = center
                        then center_cell := true
                        else adjacent_via := true
                end
        end

        /* via exclusion */
        if ( adjacent_via and not center_via )
        then via(center) := novia
end
```

APPENDIX B

LOCAL ALGORITHMS FOR GLOBAL ROUTER

This appendix presents the details of the local algorithms designed to perform global routing in an RPS pipeline. The backtrace algorithm is omitted, as we have only implemented those portions of the router that run in the pipeline. Despite the extreme simplicity of this router, the physical implementation of these local algorithms is surprisingly complex. This complexity derives from two sources. First, this router attempts to manipulate some numerical fields. This processing does not fit well with the template-match architecture of the stage hardware. Second, information about cell (x, y) appears not only in cell (x, y) but now also in its neighbors. This is because we are now processing cell *boundaries*, which are shared among cells.

In global routing, grid cells represent routing regions with boundaries that admit a maximum number of wiring tracks. In this implementation, a cell is a pair of horizontal and vertical track capacities. As shown in Fig. B.1, each cell stores channels capacities for its eastern and southern boundaries. One cell must consult its west and north neighbors to determine the other two capacities. Channels admit at most 15 tracks, or as few as 0 tracks.

The first step is to translate the initial grid with integer channel capacities into the *binary capacity* grid. In this new grid, channels with non-zero capacity are marked free, others marked blocked. This is the grid on which wavefront expansion is performed. The translation algorithm, partitioned into two subarray-computations, is as follows:

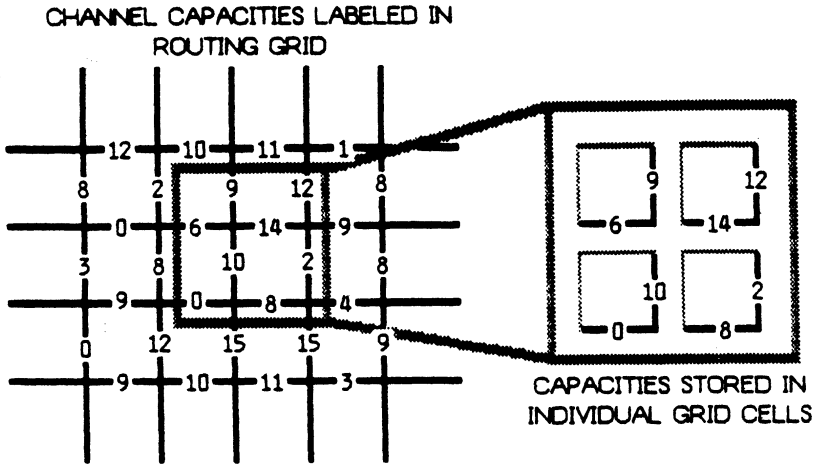


Figure B.1 Basic Cell Representation for Global Routing

Given: A 3×3 subarray of cells in a global-routing grid.

Task: Translate subarray center with non-zero channel capacities into format indicating admissible channels for wavefront expansion. The term "binary" refers to the fact that channels are either admissible or inadmissible.

Input Cell Format:

A cell is a pair indicating horizontal and vertical channel capacities:

$$\begin{aligned} \text{cell} &= (\, h\text{-}\mathit{capacity},\ v\text{-}\mathit{capacity}) \\ &\in \{0,1,\ldots,15\} \times \{0,1,\ldots,15\} \end{aligned}$$

We refer to these components of a cell as $h\text{-}\mathit{capacity}(\text{cell})$, $v\text{-}\mathit{capacity}(\text{cell})$.

Output Cell Format:

A cell is transformed into a 5-tuple indicating how a source-pointing arrow on an expanding wavefront can be attached to a channel in this cell; this indicates how a wavefront can cross this cell. The first 4 elements of the 5-tuple indicate whether a channel is free or blocked to an arrow in one of the four compass directions. The 5th element indicates whether the cell can be expanded during wavefront expansion, or holds a source-pointing arrow:

$$\begin{aligned} \text{cell} = (\, &\mathit{north}\text{-}\mathit{chan},\ \mathit{south}\text{-}\mathit{chan},\ \mathit{west}\text{-}\mathit{chan},\ \mathit{east}\text{-}\mathit{chan}, \mathit{expand}) \\ &\in \{\uparrow\mathit{free}, \uparrow\mathit{blocked}\} \times \{\downarrow\mathit{free}, \downarrow\mathit{blocked}\} \\ &\times \{\leftarrow\mathit{free}, \leftarrow\mathit{blocked}\} \times \{\rightarrow\mathit{free}, \rightarrow\mathit{blocked}\} \\ &\times \{\uparrow, \downarrow, \leftarrow, \rightarrow, \mathit{free}, \mathit{blocked}\} \end{aligned}$$

We refer to these components of a cell as $\mathit{north}\text{-}\mathit{chan}(\text{cell})$, $\mathit{expand}(\text{cell})$, etc.

```
Algorithm: Translate_to_Binary_Capacity
begin

Subarray_Computation_1:

        /* determine which channels have non-zero capacity */

        if h-capacity(center) > 0
        then east_admissible := true
        else east_admissible := false

        if h-capacity(west) > 0
        then west_admissible := true
        else west_admissible := false

Subarray_Computation_2:

        if v-capacity(center) > 0
        then south_admissible := true
        else south_admissible := false

        if v-capacity(north) > 0
        then north_admissible := true
        else north_admissible := false

        /* mark center as free or blocked to each direction
        ** of potential wavefront expansion
        */

        if east_admissible
        then east-chan(center) := →free
        else east-chan(center) := →blocked

        if west_admissible
        then west-chan(center) := ←free
        else west-chan(center) := ←blocked

        if north_admissible
        then north-chan(center) := ↑free
        else north-chan(center) := ↑blocked

        if south_admissible
        then south-chan(center) := ↓free
        else south-chan(center) := ↓blocked

        /* now that the channels are marked, the cell itself
        ** is marked free if any channel is admissible,
        ** else the cell itself is marked blocked.
        ** Only free cells can ever be expanded and given
        ** an arrow label during wavefront expansion.
        */

        if ( north-chan(center) = ↑ee ) or
                ( south-chan(center) = ↓ee ) or
                ( east-chan(center) = →ee ) or
                ( west-chan(center) = ←ee )
        then expand(cell) := free
```

else *expand*(cell) := *blocked*

end

Given the binary capacity grid, wavefront expansion proceeds by expanding source-pointing arrows on one conceptual layer, similar to the one-layer RPS router. However, now it is channels, not cells, that are free or blocked to the expanding wavefront. The limitations of current cell size preclude an explicit symbol for a source. Instead, we simply mark a channel bounding the source cell with an expansion arrow. This suffices to make the source *active* so that wavefront expansion will proceed from that point. The expansion algorithm, partitioned into two subarray-computations, is as follows:

Given: A 3×3 subarray of cells in a global-routing grid.

Task: Expand subarray center with source-pointing arrow if it has a neighbor on a wavefront.

Input Cell Format:
Same as output for format for Translate_to_Binary_Capacity. A cell is a 5-tuple, with four components indicating whether the four boundary channels are admissible, and one component indicating the status of the cell during a normal one-layer expansion: *free*, *blocked*, or ↑, ↓, ←, →. As before, we refer to these components of a cell as *north* – *channel*(cell), *expand*(cell), etc.

Output Cell Format:
Same as input format.

Algorithm: Wavefront_Expansion
begin

Subarray_Computation_1:

```
/* determine if center is free, and north, west
** neighbors are on a wavefront
*/
if ( expand(center) = free ) and
        ( ( west-chan(center) = ←free ) or
          ( north-chan(center) = ↑free ) )
then active_center := true
else active_center := false

if  expand(north) in { ↑, ↓, ←, → }
then active_north := true
else active_north := false

if  expand(west) in { ↑, ↓, ←, → }
then active_west := true
else active_west := false
```

```
        /* now choose an arrow label for center.
        ** Note: we are not guaranteed that this cell
        ** is really expandable.  We may have an
        ** active neighbor, but the channel to
        ** that neighbor may be blocked.  In that case
        ** the center cell is just marked free.
        */
        if active_center and active_north and ( north-chan(cell) = ↑free )
        then expand(cell) = ↑
        else if active_center and active_west and ( west-chan(cell) = ←free )
        then expand(cell) = ←
        else expand(cell) = free

    Subarray_Computation_2:

        /* determine if center is free, and south, east
        ** neighbors are on a wavefront
        */
        if ( expand(center) = free ) and
                ( ( east-chan(center) = →free ) or
                  ( south-chan(center) = ↓free ) )
        then active_center := true
        else active_center := false

        if  expand(south) in { ↑, ↓, ←, → }
        then active_south := true
        else active_south := false

        if  expand(east) in { ↑, ↓, ←, → }
        then active_east := true
        else active_east := false

        /* now choose an arrow label for center.
        ** Note: we are not guaranteed that this cell
        ** is really expandable.  We may have an
        ** active neighbor, but the channel to
        ** that neighbor may be blocked.  In that case
        ** the center cell is just marked free.
        */
        if active_center and active_east and east-chan(cell) = →free
        then expand(cell) = →
        else if active_center and active_south and south-chan(cell) = ↓free
        then expand(cell) = ↓
        else expand(cell) = free

    end
```

As usual, backtrace is to be done on the host. Similar to the one-layer router, backtrace appends a special symbol *trace* to wavefront arrows on a trace path. We omit discussion of this processing.

Given a traced grid, the cleanup algorithm is a translation step which produces another channel capacity grid. In this Δ-capacity grid, channels with wiring capacity diminished by one track due to the newly traced path are marked. In this new grid, each channel is marked as used or unused for the new path. Subtracting this grid from the initial routing grid gives an updated grid on which the next net can be routed. The algorithm, requiring only one subarray-computation, is as follows:

Given: a 3×3 subarray of cells in a global-routing grid.

Task: Mark channels in subarray center with capacity diminished by 1 due to traced path crossing channel.

Input Cell Format:

Similar to format for Wavefront_Expansion. Cells on backtrace path now are marked *trace*, whereas other cells are marked *notrace*. Cells are 6-tuples now, with the addition of the trace information:

$$\begin{aligned} \text{cell} = (\, &\textit{north-chan}, \textit{south-chan}, \textit{west-chan}, \textit{east-chan}, \textit{expand}) \\ &\in \{\uparrow\textit{free}, \uparrow\textit{blocked}\} \times \{\downarrow\textit{free}, \downarrow\textit{blocked}\} \\ &\times \{\leftarrow\textit{free}, \leftarrow\textit{blocked}\} \times \{\rightarrow\textit{free}, \rightarrow\textit{blocked}\} \\ &\times \{\uparrow, \downarrow, \leftarrow, \rightarrow, \textit{free}, \textit{blocked}\} \\ &\times \{\textit{trace}, \textit{notrace}\} \end{aligned}$$

We refer to these components of a cell as *north-chan*(cell), *expand*(cell), *trace*(cell), etc.

Output Cell Format:

similar to input format for Translate_to_Binary_Capacity. A cell is a 2-tuple indicating which channels have been crossed by backtrace path.

$$\begin{aligned} \text{cell} = &(\textit{h-used}, \textit{v-used}) \\ &\in \{0, 1\} \times \{0, 1\} \end{aligned}$$

We refer to these two components of a cell as *h-used*(cell) and *v-used*(cell).

Algorithm: Cleanup
begin

Subarray_Computation_1:

```
/* recall that each cell in the original
** grid with channel capacities only held
** channel info for its east and south channels.
** These channels are diminished if neighbors
** are on paths going into the center cell:
**        expand(east) = ← or expand(south) = ↑
** or if the center is on a path going into
** one of these neighbors:
**        expand(center) = → or expand(center) = ↓
```

```
        */

        h- used(center) := 0
        v- used(center) := 0

        /* first, check if neighbor cross into center */

        if ( expand(east) = ← ) and ( trace(east) = trace )
        then  h- used(center) := 1
        if ( expand(south) = ↑ ) and ( trace(south) = trace )
        then  v- used(center) := 1

        /* next, check if center crosses into these neighbors */

        if ( expand(center) = → ) and ( trace(center) = trace )
        then  h- used(center) := 1
        if ( expand(center) = ↓ ) and ( trace(center) = trace )
        then  v- used(center) := 1
end
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[AbLM82] M. Abramovici, Y. H. Levendel and P. R. Menon, "A logic simulation machine," *Proc. 19th Design Automation Conf.*, pp. 65-73, June 1982.

[Adsh82a] H. G. Adshead, "Towards VLSI complexity: The DA algorithm scaling problem: Can special DA hardware help?" *Proc. 19th Design Automation Conf.*, pp. 339-344, June 1982.

[Adsh82b] H. G. Adshead, "Employing a distributed array processor in a dedicated gate-array layout system," *Proc. ICCC*, pp. 411-414, Oct. 1982.

[Aker67] S. Akers, Jr., "A modification of Lee's path connection algorithm," *IEEE Trans. Electronic Computers*, vol. EC-16, pp. 97-98, Feb. 1967.

[Aker72] S. Akers, "Routing," in *Design Automation of Digital Systems*, vol. 1, M. Breuer, Ed., Englewood Cliffs, NJ: Prentice Hall, Chapter 6, 1972.

[Anto82] D. Antonsson et al., "PICAP - A system approach to image processing," *IEEE Trans. Computers*, vol. C-31, no. 10, pp. 997-1000, Oct. 1982.

[Aoki82] M. Aoki et al., "An LSI adaptive array processor," *Proc. ISSCC*, pp. 122-123, Feb. 1982.

[ArOu82] M. H. Arnold and J. K. Ousterhout, "Lyra: A new approach to geometric layout rule checking," *Proc. 19th Design Automation Conference*, pp.530-536, June 1982.

[Bair78] H. S. Baird, "Fast algorithms for LSI artwork analysis," *Jour. of Design Automation and Fault Tolerant Computing*, vol. 2, no. 2, May 1978, pp. 179-209.

[Bake80] C. M. Baker, *Artwork Analysis Tools for VLSI Circuits*, M.S. Thesis, MIT, Cambridge, MA, 1980.

[Bald83] J. W. Balde, "Interconnection costs will be dominant in the 80's," *Computer*, vol. 16, no. 1, pp. 83-84, Jan. 1983.

[Barn68] G. H. Barnes, "The Illiac IV computer," *IEEE Trans. Computer*, vol. C-17, no. 8, p. 746, Aug. 1968.

[Bart80] R. L. Barto, *A Computer Architecture for Logic Simulation*, Ph.D. Thesis, Univ. of Texas, May 1980.

[BaST80] R. L. Barto, S. A. Szygenda and E. W. Thompson, "Architecture for a hardware simulator," *Proc. ICCC-80*, pp. 891-893, 1980.

[Batc80] K. E. Batcher, "Architecture of a massively parallel processor," *Proc. 7th Annual Symp. on Computer Architecture*, pp. 168-174, 1980.

[BeHH80] J. L. Bentley, D. Haken and R. W. Hon, "Fast geometric algorithms for VLSI tasks," *Proc. COMPCON-80*, pp. 88-92, 1980.

[BHML82] S. E. Bello, J. L. Hoffman, R. I. McMillan and J. A. Ludwig, "VLSI hierarchical design verification," *Proc. ICCC-82*, pp. 530-533, Oct. 1982.

[BHST84] Z. Barzilai, L. M. Huisman, G. M. Silberman, D. T. Tang and L. S. Woo, "Fast pass-transistor simulation for custom MOS circuits," *Design & Test*, vol. 1, no. 1, pp. 71-81, Feb. 1984.

[Bird84] P. Bird, Private Communication, University of Michigan, May 1984.

[Blan82] T. Blank, *A Bit Map Architecture and Algorithms for Design Automation*, Ph.D. Thesis, Dept. of EE, Stanford Univ., Stanford CA., Sept. 1982.

[Blan84] T. Blank, Private Communication, June 1984.

[BLMR83] T. Burggraff, A. Love, R. Malm and A. Rudy, "The IBM Los Gatos Logic Simulation Machine Hardware," *Proc. ICCD-83*, pp. 584-587, 1983.

[BlSv81] T. Blank, M. Stefik and W. van Cleemput, "A parallel bit map processor architecture for DA algorithms," *Proc. 18th Design Automation Conf.*, pp. 837-845, June/July 1981.

[BrFI81] M. A. Breuer, A. D. Friedman and A. Iosupovicz, "A survey of the state of the art in design automation," *Computer*, vol. 14, no. 10, pp. 58-75, Oct. 1981.

[BrSh81] M. A. Breuer and K. Shamsa, "A hardware router," *Jour. of Digital Systems*, vol. IV, issue 4, pp. 393-408, 1981.

[Burk70] A. W. Burks, ed., *Essays on Cellular Automata*, Urbana:Univ. of Illinois Press, 1970.

[Carr81] C. R. Carroll, "A smart memory array processor for two layer path finding," *Proc. 2nd Caltech Conf. on Very Large Scale Integration*, Jan. 1981.

[ChBr83] D. Chyan and M. A. Breuer, "A placement algorithm for array processors," *Proc 20th Design Automation Conference*, pp. 182-188, June 1983.

[DaGe82] E. Damm, H. Gethoeffer, "Hardware support for automatic routing," *Proc. 19th Design Automation Conf.*, pp. 219-223, June 1982.

[DaLe81] P. E. Danielsson and S. Levialdi, "Computer architectures for pictorial information systems," *Computer*, vol. 14, no. 11, pp. 53-67, Nov. 1981.

[Denn82] M. M. Denneau, "The Yorktown simulation engine," *Proc. 19th Design Automation Conf.*, pp. 55-59, June 1982.

[Dona81] W. E. Donath, "Wire length distribution for placements of computer logic," *IBM J. Research and Development*, vol. 25, no. 3, pp. 152-155, May 1981.

[Duff78] M. J. B. Duff, "Review of the CLIP image processing system," *Proc. National Computer Conf.*, pp. 1055-1060, 1978.

[DuLe81] *Languages and Architectures for Image Processing*, M. Duff and S. Levialdi, Eds., London: Academic Press, 1981.

[Dunn84] L. N. Dunn, "IBM's engineering design system support for VLSI design and verification," *Design & Test*, vol. 1, no. 1, pp. 30-40, Feb. 1984.

[EuMu82] R. Eustace and A. Mukhopadhyay, "A deterministic finite automaton approach to design rule checking for VLSI," *Proc. 19th Design Automaton Conf.*, pp. 712-717, June 1982.

[FrSp81] E. H. Frank and R. F. Sproull, "Testing and debugging custom integrated circuits," *Computing Surveys*, vol. 13, no. 4, pp. 425-452, Dec. 1981.

[GiCa83] L. Gindraux and G. Catlin, "CAE station's simulators tackle 1 million gates," *Electronic Design*, pp. 127-136, Nov. 10, 1983.

[Gola65] M. J. E. Golay, *et al.*, "Apparatus for counting bi-nucleate lymphocytes in blood," U.S. Patent 3214574, Oct. 26, 1965.

[Gola69] M. J. E. Golay, "Hexagonal parallel pattern transformations," *IEEE Trans. Computers*, vol. C-18, pp. 733-740, Aug. 1969.

[Gray71] S. B. Gray, "Local properties of binary images in two dimensions," *IEEE Trans. Computers*, vol. C-20, no. 5, pp. 551-561, May 1971.

[Gris82] T. W. Griswold, "Portable design rules for bulk CMOS," *VLSI Design*, vol. III, no. 5, pp. 62-67, Sept./Oct. 1982.

[GrNE84] J. Grinberg, G. R. Nudd and R. D. Etchells, "A cellular VLSI architecture," *Computer*, vol. 17, no. 1, pp. 69-81, Jan. 1984.

[HaKu72] M. Hannan and J. M. Kurtzburg, "Placement techniques," in *Design Automation of Digital Systems*, vol. 1, M. Breuer, Ed., Englewood Cliffs, NJ: Prentice Hall, Chapter 5, 1972.

[Hell83] W. Heller, Private Communication, October 1983.

[HFPS82] J. M. Herron, J. Farley, K. Preston and H. Sellner, "A general-purpose high-speed logical transform image processor," *IEEE Trans. Computer*, vol. C-31, no. 8, pp. 795-800, Aug. 1982.

[High83] D. Hightower, "The Lee router revisited," *Proc. ICCAD-83*, pp. 136-139, Oct.-Nov. 1983.

[HKMR83] J. Howard, J. Kohn, R. Malm, and A. Rudy, "Using the IBM Los Gatos Logic Simulation Machine," *Proc. ICCD-83*, pp. 592-594, 1983.

[Hoel76] J. H. Hoel, "Some variations of Lee's algorithm," *IEEE Trans. Comput.*, vol. C-25, pp. 19-24, Jan. 1976.

[HoMW83] J. K. Howard, R. L. Malm and L. M. Warren, "Introduction to the IBM Los Gatos Logic Simulation Machine," *Proc. ICCD-83*, pp. 580-583, 1983.

[HoNa83] S. J. Hong and R. Nair, "Wire routing machines--New tools for VLSI physical design," *Proc. of the IEEE*, vol. 71, no. 1, pp. 57-65, Jan 1983.

[HoNS81] S. J. Hong, R. Nair and E. Shapiro, "A physical design machine," in *VLSI 81*, J. P. Gray, Ed., London: Academic Press, pp. 346-365, 1981.

[Ilif82] J. K. Iliffe, *Advanced Computer Design*, London: Prentice Hall, Chap. 12, 1982.

[IoKB83] A. Iosupovicz, C. King and M. A. Breuer, "A module interchange placement machine," *Proc. 20th Design Automation Conference*, pp. 171-174, June 1983.

[Iosu80] A. Iosupovicz, "Design of an iterative array maze router," *Proc. ICCC*, pp. 908-911, 1980.

[JiKo81] X. Ji-Guang and T. Kozawa, "An algorithm for searching shortest path by propagating wave fronts in four quadrants," *Proc. 18th Design Automation Conference*, pp. 29-36, 1981.

[KaSa83] R. Kane and S. Sahni, "A systolic design rule checker," TR-83-13, Computer Science Dept., University of Minnesota, July 1983.

[Kido83] M. Kidode, "Image processing machines in Japan," *Computer*, vol. 16, no. 1, pp. 68-80, Jan. 1983.

[KlSe72] J. C. Klein and J. Serra, "The Texture Analyser," *J. Microscopy*, vol. 95, No. 2, pp. 349-356, 1972.

[KMMN83] J. Kohn, R. Malm, C. Meiley and F. Nemec, "The IBM Los Gatos Logic Simulation Machine Software," *Proc. ICCD-83*, pp. 588-591, 1983.

[Kogg81] P. M. Kogge, *Architecture of Pipelined Computers*, Hemisphere Publishing Corp., 1981.

[Korn82] R. K. Korn, "An efficient variable-cost maze router, "*Proc. 19th Design Automation Conference*," pp. 425-431, 1982.

[Kung79] H. T. Kung, "Let's design algorithm's for VLSI systems," CMU-CS-79-151, Dept. of Computer Science, Carnegie Mellon University, 1979.

[Kung84] H. T. Kung, Private Communication, March 1984.

[LaRu71] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. Computers*, vol. C-20, pp. 1469-1479, 1971.

[Lee61] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Trans. on Electronic Computers*, vol. EC-10, September 1961, pp. 346-358.

[LoMc80] R. M. Lougheed, D. L. McCubbrey, "The cytocomputer: A practical pipelined image processor," *Proc. 7th Annual International Symp. on Computer Architecture*, pp. 271-277, May 1980.

[LoMS80] R. M. Lougheed, D. L. McCubbrey and S. R. Sternberg, "Cytocomputers: Architectures for parallel image processing," *Proc. IEEE Workshop on Picture Data Description and Management*, Aug. 1980.

[Losl80] P. Losleben, "Computer aided design for VLSI," in *Very Large Scale Intergration VLSI: Fundamentals and Applications*, D. F. Barbe, Ed., Springer-Verlag, 1980.

[LoTh79] P. Losleben and K. Thompson, "Topological analysis for VLSI circuits," *Proc. 16th Design Automation Conf.*, pp. 461-473, June 1979.

[Loug84] R. M. Lougheed, Private Communication, April 1984.

[Lyon81] R. F. Lyon, "Simplified Design Rules for VLSI Layouts," *Lambda*, vol. II, no. 1, pp. 54-59, 1st Quarter, 1981.

[Math75] G. Matheron, *Random Sets and Integral Geometry*, New York: John Wiley & Sons, 1975.

[McCo63] B. H. McCormick, "The Illinois pattern recognition computer--ILLIAC III,", *IEEE Trans. Electronic. Computers*, vol. EC-12, pp. 791-813, 1963.

[MeCo80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading: Addison-Wesley, 1980.

[Meye84] W. Meyer, "Hardware logic simulator alters design process," *Design & Test*, vol. 1, no .1, p. 95, Feb. 1984.

[Moor59] E. F. Moore, "Shortest path through a maze," in *Annals of the Computation Laboratory of Harvard University*, Harvard University Press, Cambridge, Mass., vol. 30, pp. 285-292, 1959.

[MRLA82] T. N. Mudge, R. A. Rutenbar, R. M. Lougheed and D. E. Atkins, "Cellular image processing techniques for VLSI circuit layout validation and routing," *Proc. 19th Design Automation Conf.*, pp. 537-543, June 1982.

[MuLT81a] T. N. Mudge, R. M. Lougheed and W. B. Teel, "Design rule checking for VLSI circuits using a cellular computer," *Abstracts of the 1981 ACM Computer Science Conf.*, St. Louis, pp. 29, Feb. 1981.

[MuLT81b] T. N. Mudge, R. M. Lougheed and W. B. Teel, "Cellular image processing techniques for checking VLSI circuit layouts," *Proc. of the 1981 Conf. on Information Sciences and Systems*, The Johns Hopkins University, pp. 315-320, March 1981.

[Nemh66] G. L. Nemhauser, *Introduction to Dynamic Programming*, New York: John Wiley and Sons, Inc., 1966.

[NHLV82] R. Nair. S. J. Hong, S. Liles and R. Villani, "Global wiring on a wire routing machine," *Proc. 19th Design Automation Conf.*, pp. 224-231, June 1982.

[NoNM81] D. Noice, J. Newkirk and R. Mathews, "A polygon package for analyzing integrated circuit designs," *VLSI Design*, vol. II, no. 3, pp. 33-36, 1981.

[Oust84] J. Ousterhout, "Corner stitching: a data-structuring technique for VLSI layout tools," *IEEE Tran. CAD ICs and Systems*, vol. CAD-3, no. 1, pp. 87-100, Jan. 1984.

[PDLN79] K. Preston, M. J. B Duff, S. Levialdi, P. Norgren and J. Toriwaki, "Basics of cellular logic with some applications in medical image processing," *Proc. of the IEEE*, vol. 67, no. 5, pp. 826-856, May 1979.

[Pfis82] G. F. Pfister, "The Yorktown simulation engine: Introduction," *Proc. 19th Design Automation Conf.*, pp. 51-54, June 1982.

[PfKr82] G. F. Pfister and E. P. Kronstadt, "Software support for the Yorktown Simulation Engine," *19th Design Automation Conference*, pp. 60-65, June 1983.

[Pres83] K. Preston, "Cellular logic computers for pattern recognition," *Computer*, vol. 16, no. 1, pp. 36-47, Jan. 1983.

[PrNo72] K. Preston and P. E. Norgren, "Interactive image processor speeds pattern recognition," *Electronics*, vol. 45, p. 89, 1972.

[PrUh82] *Multicomputers and Image Processing: Algorithms and Programs*, K. Preston and L. Uhr, Eds., New York: Academic Press, 1982.

[Rubi74] F. Rubin, "The Lee path connection algorithm," *IEEE Trans. Computer*, vol. C-23, pp. 907-914, Sept. 1974.

[RuMA83a] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "A class of cellular architectures to support physical design automation," University of Michigan, Computing Research Laboratory, CRL-TR-10-83, 1983.

[RuMA83b] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "Wire routing experiments on a raster pipeline subarray machine," *Digest of Papers, IEEE International Conf. on CAD*, pp. 135-136, Sept. 1983.

[RuMA84] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "A class of cellular architectures to support physical design automation," *IEEE Tran. CAD of ICs and Systems*, to appear 1984.

[Seil81] L. Seiler, "Special purpose hardware for design rule checking," *Proc. of the Caltech Conf. on VLSI*, Jan. 1981.

[Seil82] L. Seiler, "A hardware assisted design rule check architecture," *Proc. 19th Design Automation Conf.*, pp. 232-238, June 1982.

[Seil84] L. Seiler, *A Hardware Assisted Methodology for VLSI Design Rule Checking*, Ph.D. Thesis, MIT, 1984.

[Serr81] J. Serra, *Mathematical Morphology and Image Processing*, London: Academic Press, 1981.

[Sieg81] H. J. Siegel, et al., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers*, vol. C-30, pp. 934-947, Dec. 1981.

[SKOT83] T. Sasaki, N. Koike, K. Ohmori and K. Tomita, "HAL; A block level hardware logic simulator," *20th Design Automation Conference*, pp. 150-156, 1983.

[SlBM62] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON computer," *Proc. Western Joint Computer Conf.*, pp. 87-107, 1962.

[SNBS84] D. C. Smith, R. Noto, F. Borgini, S. S. Sharma and J. C. Werbickas, "The variable geometry automated universal array layout system (VAGUA)," *IEEE Trans. CAD of ICs and Systems*, vol. CAD-3, no. 1, pp. 20-26, Jan. 1984.

[SoRo81] J. Soukup and J. C. Royle, "On hierarchical routing," *Journal of Digital Systems*, vol. V, no. 3, pp. 265-289, 1981.

[Souk78] J. Soukup, "Fast maze router," *Proc. 15th Design Automation Conf.*, pp. 100-101, June 1978.

[Souk81] J. Soukup, "Circuit layout," *Proc. of the IEEE*, vol. 69, no. 10, pp. 1281-1304, Oct. 1981.

[Ster79a] S. R. Sternberg, "Parallel architectures for image processing", *Proc. 3rd International IEEE COMPSAC*, Chicago, 1979.

[Ster79b] S. R. Sternberg, "Automatic image processor," U. S. Patent 4167728, Sept. 11, 1979.

[Ster80a] S. R. Sternberg, "Language and architecture for parallel image processing," in *Pattern Recognition in Practice*, E. S. Gelsema and L. N. Kanal, Eds., Amsterdam: North Holland Publishing Co., 1980.

[Ster80b] S. R. Sternberg, "Cellular computers and biomedical image processing," *Proc. U.S.-France Seminar on Biomedical Image Processing*, Grenoble, France, May 1980.

[Ster83] S. R. Sternberg, "Biomedical Image Processing," *Computer*, vol. 16, no. 1, pp. 22-34, Jan. 1984.

[SzVW83] T. G. Szymanski and C. J. Van Wyk, "Space efficient algorithms for VLSI artwork analysis," *Proc. 20th Design Automation Conference*, pp. 734-739, June 1983.

[TYKS80] F. Tada, K. Yoshimura, T. Kagata and T. Shirkawa, "A fast maze router with iterative use of variable search space restriction," *Proc. 17th Design Automation Conference*, pp. 250-254, 1980.

[UeKH83] K. Ueda, T. Komatsubara and T. Hosaka, "A parallel processing approach for logic module placement," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-2, no. 1, pp. 39-47, Jan. 1983.

[Ulam70] S. M. Ulam, in *Essays on Cellular Automata*, Chaps. 5-6, Urbana:Univ. of Illinois Press, 1970.

[vanA71] A. W. Van Ausdal, "Use of the Boeing computer simulator for logic design confirmation and failure diagnostic programs," *Advanced Astronautics Science*, 29, pp. 573-594, June 1971.

[vanB83] N. van Brunt, "The ZYCAD logic evaluator and its application to modern system design," *Proc. ICCD-83*, pp. 232-233, 1983.

[WeLF82] C. Weems, S. Levitan, and C. Foster, "Titanic: a VLSI based content addressable parallel array processor," *Proc. ICCC-82*, pp. 236-239, 1982.

[Wilm80] J. Wilmore, "The use of bit maps in designing efficient data bases for integrated circuit layout systems," *Jour. of Digital Systems*, vol. IV, issue 1, pp. 71-95, 1980.

[Wils83] G. L. Wilson, "1983 Spring IEEE/ACM Design Automation Workshop," *ACM SIGDA Newsletter*, vol. 13, no. 2, pp. 4-11, July 1983.