A DISTRIBUTED LAYOUT COMPACTOR

by
RODRIGUE BYRNE
B.Sc(Hons), B. Eng, Memorial University of Newfoundland, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIRMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department of
Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Supervisor Dr. G.C. Shoja

[redacted]
Dr. W.W. Wadge

[redacted]
Dr. F. El Guibaly

[redacted]
Dr. N.J. Dimopoulos

ISBN 0-315-46509-3

Supervisor: Dr. G. C. Shoja

Abstract

A distributed algorithm used to solve the symbolic compaction problem, running on a set of computers attached to a local area network, is presented in this thesis. The symbolic compaction problem consists of translating a symbolic description of a VLSI layout into the smallest possible mask level description without introducing any design rule violations.

The compaction algorithm is based on a virtual grid approach similar to the ones used in the MULGA and VIVID systems. The serial version of the compaction algorithm is shown to have linear time complexity with respect to the number of layout primitives. Also for the benchmarks performed the compactor was 9 to 14 times faster than VIVID's compactor.

The distributed algorithm developed follows the client/server model of distributed systems. The client is responsible for partitioning the layout problem into separate regions to be compacted by a set of server processes, and for mergering these separately compacted regions into the final mask descriptions. The Distributed Layout Compaction System (DLCS) thus developed was tested on three different hardware configurations with input layouts ranging in complexity from simple NAND gates to a static memory array. The benchmark results were broken up into communication time, serial time, and parallel time categories. The serial aspect of the distributed compaction algorithm required on average 40% of the real time taken to solve a problem. Thus the serial part of the algorithm limits the speed up to 2.5. The conclusion of the thesis proposes some ways of enhancing the performance of this type of distributed compaction system.

Examiners: [redacted]

Supervisor Dr. G.C. Shoja

[redacted]

Dr. W.W. Wadge

[redacted]

Dr. F. El Guibaly

[redacted]

Dr. N.J. Dimopoulos

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor, Dr. G. C. Shoja, for his patience and help during the research and writing of this thesis. The following people, from Canada's east and west coasts, provided helpful comments during the writing of this thesis: Elaine Boone, Paul Gillard, Sheila Singleton, and Micaela Serra. I would like to acknowledge the Computer Science Department at Memorial for allowing me access to their computing systems for testing the compaction system and typesetting of the thesis. Finally, none of this work would have been possible without financial assistance from the University of Victoria's Computer Science Department and NSERC.

DEDICATION

To my Mom, *Helen Byrne.*

Chapter 1

Introduction

1.1 A Distributed Layout Compaction System

This thesis addresses the question

> *"How can the problem of layout compaction be solved in a distributed manner on a set of workstations connected by a local area network?"*

The goals of the thesis are therefore two-fold: 1) to develop a parallel algorithm which solves the layout compaction problem and 2) to implement and study this algorithm in a workstation environment. The motivation for this work comes from the observation that one of the requirements of a symbolic layout system is to provide the IC designer with quick translation from the symbolic to the mask level. The choice to base distributed compaction on a workstation environment comes from the fact that networked workstations are becoming one of the standard design environments and that there is usually excess processing power located somewhere in the network. It was also hoped that the process of implementing one distributed algorithm on a network of workstations will provide some insights into the types of

system services (i.e. runtime support) required for efficient implementations.

Layout compaction is a VLSI design tool used in the synthesis of mask descriptions for integrated-circuits. The input to a compactor usually consists of a symbolic description of the circuit, but it can consist of a mask level description. The idea behind a compactor is to remove all unnecessary space from a layout by moving all layout primitives to their closest permissible distances. Before a distributed compactor was designed a study of the algorithms used for compaction was performed. Chapter 2 contains a summary of the compaction techniques found in the literature. The virtual grid approach to compaction was chosen as the best one to transform into a distributed algorithm. It was chosen primarily because the virtual grid could be partitioned into sub-regions without having to examine the layout. The symbolic language for design description was based on the ABCD language used in the VIVID system. A BNF description of the language is given in Appendix A.

The Distributed Layout Compaction System (DLCS) consists of approximately 10,000 lines of C-language code, including comments. The system runs on VAX's and SUN's running 4.2/3UNIX [1] and Versions 3.2/4 of SUN's implementation of 4.2/3BSD UNIX. The initial technology supported was CMOS1B provided by Northern Telecom. Interprocess communication was accomplished with the use of SUN's Remote Procedure Call (RPC) system. The last part of Chapter 2 discusses the interprocess facilities provided under UNIX. The system has been tested on networks composed of SUN 3's, a VAX11/780, a VAX11/750 and microVAX-II's.

Most of the design issues and the details of the compaction algorithm are considered in Chapter 3. The overriding concern in the design of the algorithms was runtime efficiency because of the large problem sizes typically encountered in VLSI

[1] UNIX is a trademark of AT&T Technologies, Inc.

CAD. The other major concern was designing the steps of the compactor so that they could easily be partitioned among a set of processors. In the DLCS system the compaction problem was broken up into the following steps: parsing, plotting, netlist extraction, x-compaction, y-compaction, and fleshing. The parsing step converts the ABCD layout description into an internal representation. Plotting and netlist extraction steps transform the symbolic description into one where the spacing calculation needed for compaction can be performed efficiently. Compaction is performed in two phases: a one-dimensional compaction which removes excess space along the x-axis, and a one-dimensional compaction along the y-axis. The x and y compactions yield the physical position of the virtual grid. The grid position and the symbolic description are used by the fleshing step to produce the mask layout.

The transformation of the serial compaction algorithm described in Chapter 3 to a distributed version is considered in Chapter 4. The major goal of the distributed compactor is to realize a linear decrease in execution time with a linear increase in the number of processors working on the problem. The approach used to achieve parallel execution of the compactor is based on the fact that the spacing calculation of local areas of the layout only requires information from that local area. This observation leads to a partitioning of the layout into regions which will be both compacted separately and in parallel. After the separate regions are compacted, they are merged to form the mask layout.

The distributed system developed follows the client/server model of distributed systems. One process, the client, parses the symbolic description and assigns the separate regions to server processes. The server processes then perform x-compaction on their assigned regions. The x-compaction results are then communicated back to the client, which merges the x-grid spacing and assigns the

y-compaction step to the servers. The servers then perform the y-compaction and transmit their results to the client. Finally the client produces the mask layout with the fleshing step. The SOCKET STREAM style of inter-process communication was used in DLCS. This style provides reliable communication between exactly two processors. This style of communication does, however, cause a performance loss in the system. Berkeley UNIX provides another style of communication service, DATAGRAMS, which allow one to many communication but at the price of unreliable data delivery. Section 4.5 outlines the main procedures used to implement DLCS.

The DLCS was evaluated for a range of test cases and hardware configurations. The results and test methods are presented in Chapter 5. Cell layouts ranging in complexity from four-input NAND gates to a static memory array were used as input test cases. Measurements of the compactor's timings were obtained with UNIX's processing accounting system calls. The amount of real time and time spent in actual execution was measured for the client and all the servers for each test case. The timing results were separated into serial, parallel, and communication categories. The serial time measures the parts of the algorithm which were spent in strictly sequential execution. Communication timings measure the time spent in sending information between the separate processors of the system. The parallel category, of course, measures the amount of time the DLCS was performing useful work in parallel. To provide a benchmark against which the distributed results could be measured, a serial version of the DLCS and VIVID's HCOMPACT were run on the test cases for each of the different processor types. These serial benchmarks also provided a measure of the processing power difference between the different processors.

The three hardware configurations tested were:

1. a VAX11/750 as the client and three microVAX's as servers.
2. a Sun 3/160 as the client and three microVAX's as servers.
3. a Sun 3/50 as the client and six Sun 3/50's as servers.

Configuration 3 provided the best distributed results. The worst results were obtained from the second configuration, which matched a fast client with slower servers. The results for the second configuration show that care must be taken in algorithm design and processor assignment when the processors available in the network have significantly differing processing powers. Tables 5.4, 5.5, and 5.6 summarize the timing measurements for the three configurations by giving the best distributed time, the serial time, and speed up ratio for each test case. For the three configurations, Appendices B, C, and D, contain tables of the timing data separated into the serial, parallel, and communication categories for the individual test cases. Data from Appendix D was used to construct the plots in Figures 5.3 and 5.4, which depict the runtime behaviour for each test case input for the configuration of the seven Sun 3's as the number of processors were varied. The general trend observed is that, as the number of processors increases, the time the user waits for a compaction initially decreases and then starts to increase. This behaviour can be explained by noting that the communication times increase as the number of server processors increase, and this added communication time eventually counteracts any time saved in parallel computation.

The conclusions of the thesis, Chapter 6, can be separated into those dealing with just the compaction algorithm and those dealing with the distribution of the algorithm using the runtime support available on the workstations. In comparison with VIVID's HCOMPACT 1.2, the serial version of the compactor was 9 to 14 times

faster. The current serial compactor has some shortcomings which are addressed in Chapter 6, as well as some suggested directions to follow to overcome these shortcomings.

The serial aspect of the distributed compaction algorithm required, on average 40% of the real time taken to solve a problem. Thus the serial part of the algorithm limits the speed up to 2.5. The serial part includes the parsing and fleshing steps and some of the communication times. To achieve further improvements attention must be paid to decreasing the fraction of time spent in the serial computations. One facility not provided by the runtime support, which would have increased the amount of parallelism, was a reliable broadcast mechanism from one process to a set of processes. A reliable broadcast service would have reduced the amount of time spent in distributing the layout description to the servers.

Chapter 2

Background on Layout Compaction

2.1 Introduction

Compaction is the term given to the process of reducing the space between circuit elements in the layout of an integrated circuit without modifying the topology. The topology is the relative placement of components with respect to each other. The process of compaction is usually associated with a symbolic description of the layout. To produce the mask layout, the symbolic description must be translated into the mask layers and compacted to the smallest possible layout without introducing design rule errors. Compaction can also be used at the mask level, especially in the conversion of one technology to another [1]. It is the responsibility of the mask generation step to ensure that the mask layout is constructed with no design rule violations. The compaction process ensures that none of the spacing design rules are violated.

In the compaction literature the four main areas of research are 1) Virtual Grid, 2) Constraint Graph, 3) Shear Line, and 4) 2-D techniques. The first three areas solve the compaction problem in two steps: 1) compact the layout in the horizontal direction, and 2) compact the layout in the vertical direction. It has been shown by [2] that compaction in both direction is NP-complete. By performing the compaction in one dimension at a time an optimal compaction is not guaranteed but the time complexity of the process is polynomial. The last area, 2-D compaction, deals with algorithms which perform compaction in both directions, but can require exponential time. An overview and classification of compaction algorithms is found in [3]. A tutorial style discussion of compaction by Wolf can be found in [4]. An experimental comparison of one-dimensional algorithms is also performed by Wolf [5]. He states that the comparison the constraint graph approaches produce more compacted layouts than the virtual grid approach but they also take 5 to 20 times longer. A survey paper by Cho [6] gives a subjective review of compaction, concentrating on the constraint graph approach.

2.2 Virtual Grid Compaction

In the virtual grid approach the symbolic layout is embedded in a grid composed of vertical and horizontal grid lines. Symbols representing the layout primitives for devices, wires and contacts are placed on the grid points. Devices and wires can occupy more than one grid point. Compaction using a virtual grid approach is achieved by moving the vertical and horizontal grid lines as close together as possible.

The CAD systems VIVID [7] and MULGA [8,9,10] use virtual grid compaction

to generate the mask layout from their symbolic descriptions. In MULGA, ICDL (Intermediate Circuit Description Language) is the name of the symbolic layout language. VIVID calls its symbolic language ABCD (A Better Circuit Description). These languages specify the placement of symbols representing circuit elements on a virtual grid. Figure 2.1 shows a plot of a typical symbolic layout of a CMOS inverter.

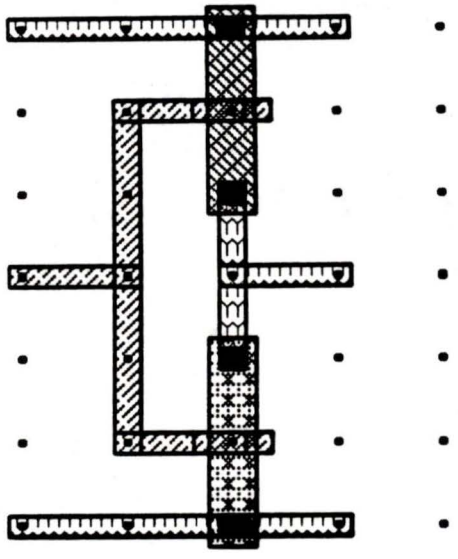


Figure 2.1: A CMOS Inverter

The layout is compacted by determining the minimum spacing between grid lines in the horizontal and vertical directions. The spacing of the grid lines is determined in two steps. First, the layout is compacted horizontally (X-compaction) by pushing all the vertical grid lines as far left as possible. Next, vertical compaction (Y-compaction) is accomplished by pushing the horizontal grid lines downward as far as possible. A more compact layout can sometimes be generated by performing a sequence of horizontal and vertical compaction steps.

The spacing of two grid lines is determined by scanning each point along the length of the grid lines and calculating the maximum spacing for each pair of points.

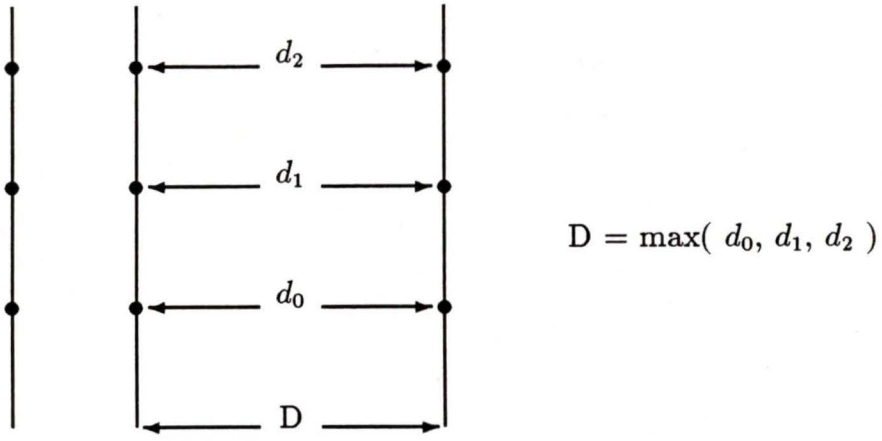


Figure 2.2: Grid Lines

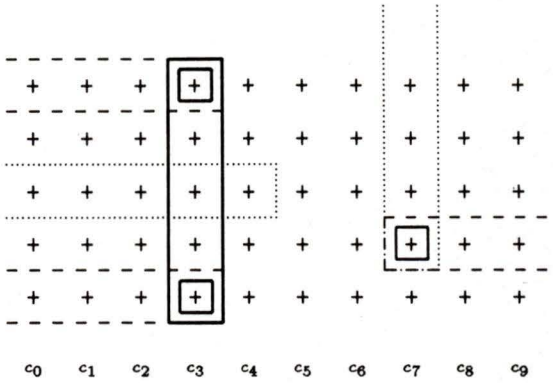
A point on the grid is defined by the intersection of two grid lines. Figure 2.2 illustrates the spacing calculations for a pair of columns. For the X-compaction only, the spacing of the horizontally adjacent points are the only ones which can be considered. The diagonally adjacent points can not be consider since the spacing between the horizontal grid lines has not been determined and will not be determined until the Y-compaction step. In Y-compaction the spacing of all the points which could cause design rule violation are considered. The spacing of two adjacent grid lines can be affected by the the contents of the surrounding grid lines. This means that in the spacing calculation of grid lines the preceding grid lines have to be examined. These added checks can be eliminated by maintaining a **frontier** in the compaction direction as the design is compacted. This frontier records the most recent position [11] of each layer for each point on the grid line. The frontier can be thought of as an outline tracing of all the mask features visible from the current grid line.

Although the input description for the two systems, MULGA and VIVID, is

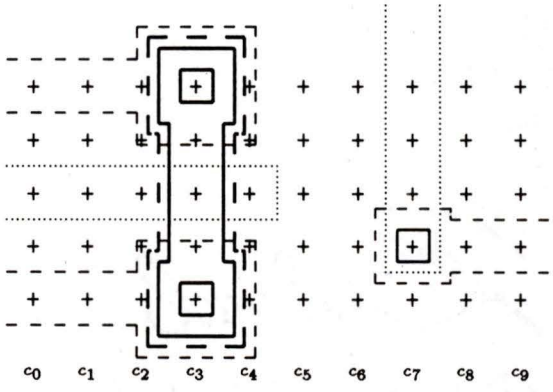
almost identical, the two systems take a different approach in the way they perform the compaction. Fleshing is the term used to describe the translation of a symbol to the mask layers used to implement that symbol. MULGA translates a symbolic layout into a mask layout by first determining the spacing before it performs the fleshing. VIVID does these operations in reverse; it first performs fleshing, then spacing. Details of the compaction process for both systems are given in sections 2.2.2 and 2.2.3.

Figure 2.3 shows the typical steps involved in a virtual grid compaction. The symbolic layout of a device connected to three wires from the left and a joining of two wires to the right of the device is shown in Figure 2.3a). The +'s in the figure show the virtual grid points. Part b, shows the symbols translated into the their respective mask descriptions. After this transformation, the virtual grid compactor can determine the spacing between the vertical columns. Since there are no symbolic layout primitives between columns c_5 and c_6, and thus no mask layers, the spacing between c_5 and c_6 is set to 0. Finally in Figure 2.3c) the horizontally compacted layout is shown. To produce the final layout the vertical compaction must be performed.

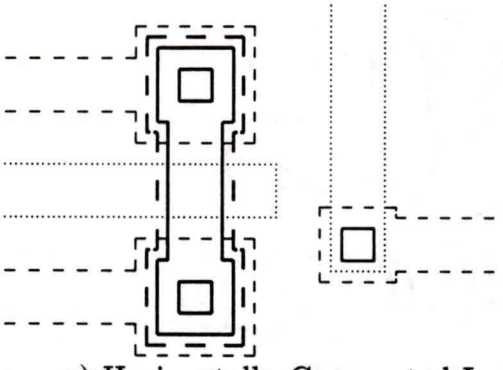
One major advantage that the virtual grid compaction approach has over the other 1-D compaction approaches is that there is no need for artwork analysis. This analysis, similar to that performed by design rule checkers, does not have to be performed because the virtual grid defines the relative placement of and connections of all the symbols. The absence of artwork analysis allows the compaction algorithm to have time complexity $O(n)$ where n is the number of symbols. With artwork checking the algorithm has time complexity at least $O(n \log_2(n))$. The time complexity increases because the algorithm has to determine the relationship



a) Symbolic Layout



b) Fleshed Layout



c) Horizontally Compacted Layout

Figure 2.3: Virtual Grid Compaction Example

between the circuit elements. For a virtual grid system the relationship between circuit elements is implicit in their placement in the grid. The $O(n \log_2(n))$ can be achieved with a line scanning algorithm [12].

2.2.1 ICDL - MULGA design language

ICDL is the input language, accepted by the MULGA system compactor, used to specify the symbolic layout of the cells. Since ABCD, the design language for VIVID, is similar to ICDL it will not be described here. To discuss ICDL, an example describing a CMOS inverter is given in Figure 2.4. Statements 1 and 16 enclose the definition of a cell. The word following **begin** on line 1 is the name of the cell. Each of the remaining lines describes the characteristics and placement of the symbols which make up the cell. The first element on a line is the type of symbol. The possible types are **dev** for device, **con** for contact, **wire** for wire, **inst** for instance, and **pin** for pin. Only the first three symbols describe circuit elements. The remaining strings on the line are parameters for the symbol and pairs of integers giving the virtual grid coordinates for the symbol. An example of an n-device is given on line 14 and a p-device on line 15. The two parameters following **dev** give the type (n or p) and orientation of the device. Finally, a pair of integers gives the virtual grid location for the gate. The definition of a wire includes the layer and a list of pairs of integers giving the path of the wire. Lines 10 to 13 are the poly wires which connect the two transistor gates together. The source and drain of the p and n transistors are connected by the metal wire on line 9. Finally, lines 7 and 8 are the contact cuts connecting the metal to the diffusion of the two transistors. The parameters for **con** give the type and location of the contact. The other metal lines connect VSS and VDD to transistors source and drain.

```
 1 begin inverter
 2       con         cut 2 6
 3       con         cut 2 0
 4       wire        metal 0 6 3 6
 5       wire        metal 0 0 3 0
 6       wire        metal 2 3 3 3
 7       con         cut 2 4
 8       con         cut 2 2
 9       wire        metal 2 2 2 4
10       wire        poly 0 3 1 3
11       wire        poly 1 5 2 5
12       wire        poly 1 1 1 5
13       wire        poly 2 1 1 1
14       dev         p or=1 2 5
15       dev         n or=1 2 1
16 end inverter
```

Figure 2.4: ICDL - Inverter

When the individual circuit symbols are embedded in the grid they occupy a varying number of grid points. The devices occupy three points, one each for the source, drain and gate. A contact occupies one point, while a wire occupies all the grid points it spans. A wire must span at least two points, its start and end.

2.2.2 Compaction Algorithm used by MULGA

The compaction algorithm used by MULGA is composed of spacing and fleshing steps. The output of the spacing step is the spacing for each horizontal and vertical line. The original ICDL description and grid spacing are used in the fleshing step to produce the mask file.

Spacing is performed in two passes, one pass for the horizontal lines and one for the vertical lines. A matrix is used to access the symbols placed on the grid. Each entry in the matrix corresponds to a point on the grid. An entry consists of

6	m	m	m,C,S	m
5		p	p,G	
4		p	m,C,D	
3	p	p	m	m
2		p	n,C,S	
1		p	p,G	
0	m	m	m,C,D	m
	0	1	2	3

Figure 2.5: Plotted Matrix

a record of pointers to the possible symbols on the corresponding grid point. The compaction begins by plotting the ICDL description on a matrix. Figure 2.5 shows the plot of the CMOS inverter, where m is a metal wire, p is a poly wire, C is a contact, S is the source of a device, G is the gate of a device, and D is the drain of a device.

An entry is plotted for each grid point the symbol occupies. With the exception of wires all other symbols are completely plotted. For horizontal wires and horizontal compaction, only the end points of the horizontal wires are plotted. This permits wires to be stretched or shrunk. The same applies for vertical wires during vertical compaction.

After the matrix has been plotted for horizontal compaction each pair of adjacent columns is examined. Only the column entries in the same row are examined for spacing constraints. For each entry the maximum width of the mask generated for the entry's symbol is calculated. For example, if the entry points to a metal wire then the width of that wire is calculated. The MULGA system assumes that the masks for the symbols are centered on the grid line. The spacing constraint due to

two entries, a and b, is calculated by:

$$(W_a + W_b)/2 + S_{ab}$$

where W_a is the width of the worst layer at a, W_b is the width of the worst layer at b and S_{ab} is the worst spacing rule for the layers at a and b. The spacing of the grid lines is the maximum of all the spacing constraints. Sometimes MULGA has to check more than two adjacent points to determine the spacing constraint. For horizontal compaction some of the points to the left of the current points have to be examined. [11] gives a method to avoid this backtracking by maintaining a **frontier.** A frontier is a data structure which records the distance of the nearest mask layer to the current grid line under consideration. As the grid is scanned the frontier is updated to reflect the addition of mask layers and the spacing of those mask layers to the current grid line.

The steps for vertical compaction are exactly the same except that pairs of rows are scanned to determine the vertical grid spacing. More than just the entries in the same column have to be examined to account for diagonal design rules.

The fleshing operation consists of generating the layout masks from the ICDL description and the vertical and horizontal grid spacing.

2.2.3 HCOMPACT

The VIVID compactor, HCOMPACT (Hierarchical Compact) [13], first compacts leaf cells individually and then pitch matches these leaf cells to generate the final mask layout. A leaf cell is composed of only primitive layout symbols. Pitch matching is the process of aligning the two leaf cells so that the connections required between the two cells are made. This process usually requires the stretching of one

of the cells. In HCOMPACT cell compaction, the fleshing step is performed before the spacing step.

In the fleshing step the ABCD symbols are translated into virtual mask rectangles (VMR rectangles). Each rectangle is associated with a single virtual grid point. A matrix is constructed containing the list of rectangles present at each corresponding grid point. Since the compactor uses the VMR rectangles to determine the grid spacing, some high-level information must be associated with each rectangle. A mask layer, a group identity number and an electric net number constitute all the high-level information associated with a rectangle. The mask layer identifies the particular process mask (e.g. polysilicon, diffusion). Since the spacing of VMR rectangles making up a transistor is different from the spacing of other rectangles, a unique group identity number is assigned to the rectangle which comprises a transistor. Finally, if two rectangles are part of the same electrical node then the spacing design rules can be ignored. Because of this approach, a wire must be broken up into a set of rectangles, with one rectangle for every virtual grid point that the wire spans.

The spacing step is divided into horizontal and vertical compaction phases. Like the approach used in MULGA for horizontal compaction, two adjacent vertical grid lines are examined. Only the points sharing a common horizontal line are compared. The maximum spacing which avoids violating design rules for the two sets of VMR rectangles is calculated. The spacing is then the sum of half the widths of the two worst VMR rectangles from each point added to the worst design rule spacing. By maintaining a frontier of the mask layers present in the preceding grid lines the mask features of the preceding lines do not have to be examined to determine the spacing of the current grid line. The vertical compaction is similar except that

more than the adjacent grid points have to be compared to avoid corner design rule violations.

Since the cells have to eventually be recombined, HCOMPACT has to examine the connection environment for each cell. Connections between cells are only allowed along a cell boundary. The boundary between two cells can occur at one grid line. The symbols placed on that grid line by the two cells influence the spacing of the adjacent grid lines in the two cells. HCOMPACT determines the worst environment for a cell and uses that environment to determine the spacing to the edge of the cell. If two cells share an empty grid line, then by convention each cell will set the spacing of that grid line as half the maximum space allowed by the design rules. These conventions for performing the spacing calculation allow a more time efficient algorithm at the expense of sometimes increasing the area of the layout.

The next step HCOMPACT takes is to join together the different cells. Before the final mask layout is produced, a process of anti-feature elimination must be performed. An anti-feature occurs when two polygons of the same type are positioned closer than is allowed by the design rules because they have the same electric node or form part of the same structure. The space between the two polygons is the anti-feature. The anti-feature eliminator fills in this space with the correct layer. The compaction process is now complete.

2.3 Shear Line Approach

The shear line approach [14-16] to compaction works by removing unnecessary space from the layout. It removes space from the X and Y directions in separate phases and thus is a 1-D compaction technique.

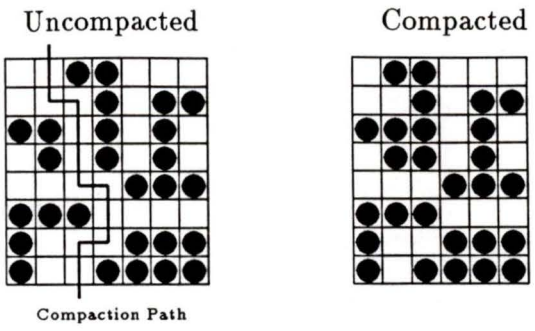


Figure 2.6: Example of Shear Line Compaction

In [14] the shear line approach is implemented on a coarse grid. A square in the grid contains either some fixed-size mask elements or no mask elements. Wires (connections between fixed size mask elements) can cross a square with no fixed mask elements. The layout is compacted by first scanning the matrix in the horizontal direction looking for "compression ridges." When a ridge is found the entries covered by that ridge are removed. Figure 2.6 shows a coarse grid before and after a ridge has been found and removed. The horizontal lines in the compaction path are examples of **shear** lines. Shear lines are perpendicular to the compression ridge and are used to shear the compression ridge, left and right, in search for a complete path across the layout. After the horizontal compaction, a vertical compaction is performed in exactly the same way. This process is repeated until no other compression ridges can be found.

A compression ridge has the following three properties:

1. It must extend from one side of the layout to the opposite side;
2. All the connections that it crosses must be perpendicular to it;
3. It cannot cross an entry with a fixed mask element.

If the compression ridge could not extend across the layout then shear lines could be used to move the ridge to a new row or column. A shear line must:

1. begin and end at the ends of a compression ridge's segment.
2. not cross any connection parallel to the compression ridge.
3. not cross any entries which contain fixed masks layouts.

Dunlop's SLIM [17] also uses the shear line approach. This compaction algorithm is broken into local and global compaction. SLIM produces a mask layout in several steps. Again horizontal and vertical compactions are performed in separate steps. First an initial mask layout from the symbolic description is generated. The initial layout contains no design rule violations. Dunlop's paper uses the term "atomics" to refer to the mask descriptions generated from a symbol. Assuming vertical compaction, atomics with the same y-coordinate are grouped into rows. A constraint graph (see next section) is then constructed for all atomics. Using this graph the compactor then performs a critical path analysis to determine where local compaction can be performed. Local compaction uses the critical path information to move all the excess space to one end of the path. Jog insertion also uses the critical path data to determine jog points. A jog is a point on a wire where the wire can be split into two wires connected at the jog point by a stretchable wire.

Global compaction first removes the excess space from the end of the critical path. It then searches for compression ridges in the layout. Unlike [14], a coarse grid is not used. The definitions for shear lines and compression ridges are the same as [14] except for modifications needed to account for the fact that a coarse grid is not used. Compaction is complete after several iterations of horizontal and vertical compaction.

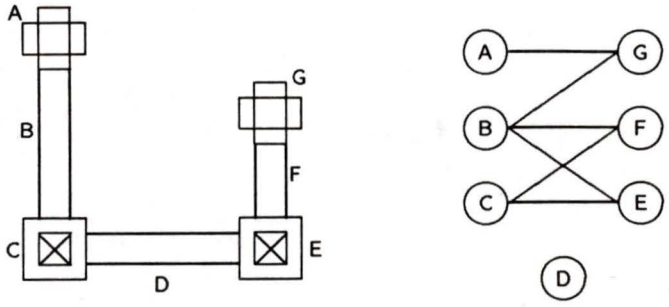


Figure 2.7: Layout and Constraint Graph

2.4 Constraint Graph Compaction

The constraint graph approach [6,18-27] to compaction has been the approach most widely considered in the literature. In this approach the symbols, or fixed mask features of a layout, are considered to be nodes in a graph. The required separation between the primitives of the layout is specified by placing a weighted arc between the associated nodes in the graph. Figure 2.7 [28, page 125] shows a layout and the corresponding constraint graph. Symbols A and G correspond to devices, symbols B, D, and F to wires and C and E to contacts. Arcs are only placed between nodes where the mask layers produced for that node are physically adjacent in the current compaction direction.

In some of the papers, a "sticks" [29,30] like symbolic description is assumed. The sticks description uses sticks as wires and nodes as transistors or contacts. The sticks are drawn parallel to the x- or y-axis. The description is concerned with the relative placement of the layout, not the specific physical dimension. In this compaction technique all objects which share the same x or y coordinates, depending on compactions direction, are considered as one group and are not separated by the

compaction process.

There are two main steps in the constraint graph approach: 1) build the constraint graph by a design rule analysis of the components and 2) solve the resulting graph using a longest path method to minimize chip area. The solution of the graph problem gives the physical placement of the symbols from the leftmost or topmost part of the layout.

For compaction in the X direction an arc is added to the constraint graph between an object and all the objects which lie directly to the right of it. The arc is not added between two objects if there are intervening objects. This technique is called **shadowing** [24]. The arc represents an inequality of the form $x_i - x_j \geq k$, where component i has to be k units to the right of component j. The units could be in microns or lambda units. Dummy components are usually placed as boundary elements at the leftmost and rightmost points in the constraint graph. In horizontal compaction either of the components are used as the boundary (i.e. the component that all others are pushed towards). The scan-line approach, used by Design Rule Checkers (DRCs), can be modified to build this graph.

One of the problems of the constraint graph approach [18] is the tendency to sometimes increase the overall wire length. This problem can be solved by adding constraints which specify both the minimum and maximum separation distances for the critical layout regions.

To find the leftmost position of each component the longest path between it and the dummy component is found. The weight of this path gives the x coordinate of the component. For acyclic graphs a depth first search can find all the longest paths. If the graph contains cycles then a modified shortest path algorithm can be used.

2.5 2-D

The 2-D approach [31], [2], [32], [31], [33], [34], [35] to compaction attempts to compact the layout in both directions simultaneously. This approach is theoretically appealing but has been shown to be NP-Complete [2]. To avoid the exponential time complexity, some of the approaches use 1-D compaction but also look at 2-D features to optimize the compaction.

One proposed technique [2,32] for 2-D compactions is to generate a totally collapsed layout and then remove the distance violations one by one until a legal configuration is reached. A branch and bound algorithm is used to remove the violations. The search tree can be efficiently pruned in many ways. Constraint graphs for the horizontal and vertical direction are the input to the algorithm.

In another algorithm the 2-D compaction problem is stated as a minimization problem; that is *find the minimum of the area function subject to linear and nonlinear constraints.* The compaction problem is formulated as a mixed integer-linear programming problem. A graph-based optimization algorithm is used to solve the resulting problem. This algorithm, still exponential in the worst case, uses horizontal and vertical constraint graphs.

In [34] a modification of the standard 1-D constraint graph approach is proposed to allow the designer to control the cell size in one direction. The standard 1-D compaction algorithms cannot recognize whether a small rearrangement of some of the components in one dimension will significantly reduce the cell size in the perpendicular direction. An example of this type of x-y interlock is given in Figure 2.8. If object A or B is moved vertically before the horizontal compaction step, then the resulting layout will be smaller.

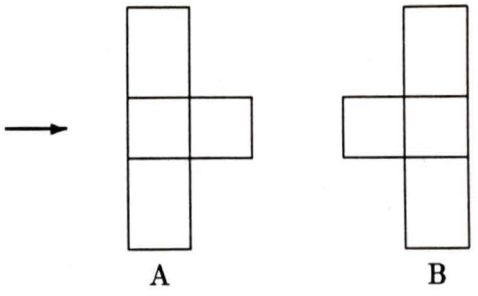


Figure 2.8: An Example of a x-y Interlock

To optimize the size of the cell in a preferred direction the following steps are performed repeatedly until no further improvement can be realized:

1. Examine the critical path graph to choose the components to be moved.
2. Shear each component pair.
3. Improve the cell compaction in the preferred dimension.

The nodes in the critical path graph are the objects in the cell. Two special nodes, LL and UR, are added to the bottom and top of the cell. Two nodes are connected by an edge if the two components block each other in the preferred dimension. Each edge defines a pair of components which may be reorganized. To reorganize the cell a set of edges must be found with the following properties:

1. They form a cutset of the graph.
2. The set separates the LL and UR nodes.
3. Each edge is cuttable; the components on the edge move closer together in the preferred dimension when reorganized.

This set of edges is used to determine how to modify the constraint graph to give a smaller size in the preferred direction.

2.6 Choice of Compaction Algorithm

This section will examine the four approaches to compaction with respect to deriving a distributed algorithm (DA). A common problem with all VLSI CAD algorithms is the efficient handling of the large number of components that typically compose the algorithm's input. Even with polynomial time algorithms, the amount of CPU-time required can be quite large. If a suitable DA could be developed then the time spent waiting for compaction could be reduced.

In order for a successful DA to be developed the problem must be divisible into sub-problems which can be performed in parallel. The results of the sub-problems must then be combined to give the solution to the entire problem. The additional work of partitioning and merging required by a DA should be compensated by the parallel execution of parts of the compaction algorithm.

One approach to distributing the compaction problem is to divide the layout into separate regions which would be compacted individually and then merge the regions to give the compacted layout. The sub-division of the layout should meet the following goals:

1. Nearly equal amounts of work should be assigned to each processor (assuming equally powerful processors).
2. The sub-division should be chosen such that the amount of work needed to merge the sub-regions is minimized.
3. The resulting layout should not be larger than the one produced by a non-distributed algorithm.
4. The partitioning process should be as simple as possible (ideally no examination of the entire layout).

The next four subsections will examine the four approaches to compaction in

light of these goals to see which one would be a candidate for a distributed algorithm implementation.

2.6.1 Virtual Grid

A distributed algorithm based on virtual grid compaction is possible by subdividing the layout. Since the layout in the virtual grid approach is represented as a rectangular grid a possible sub-division is to slice the layout into uniform strips perpendicular to the direction of compaction. This partitioning allows the spacing calculation of the grid lines in each strip to be performed in parallel. Since there are two compaction directions the layout has to be subdivided twice, where one compaction step must precede the other.

This method of transforming the virtual grid algorithm to a distributed algorithm achieves all the goals outlined in section 2.6 with the following qualifications. To ensure that roughly equal amounts of work are assigned to each processor the layout should have a uniform density of component placement. The assumption about uniform density seems reasonable since most designers will be trying to achieve the densest possible layout. If the bounding box of the layout is precalculated then the layout can be divided without examining the layout. The bounding box can be calculated when the symbolic description is parsed. When the grid is subdivided, sometimes a component can be split across two partitions; the merging process must handle this case correctly. When the strips are merged to produce the final layout, care must be taken to ensure that the spacing between strips is as small as possible.

2.6.2 Constraint Graph

The graph generation phase of the constraint graph approach can be adapted to a distributed algorithm by dividing the layout into strips and generating the constraint graph for each strip. A parallel design rule checker [36] was implemented using this strategy and since constraint graph generation is the same this technique should work. The layout would have to be partitioned twice, once for the x constraint graph and once for the y constraint graph. The merging of adjacent regions to piece together the constraint graph can entail an examination of a large portion of the two strips. Care must be taken to ensure that this examination is limited, or the merging operation can become too expensive. Most systems which implement the constraint graph compaction use a coarse grid coordinate system (although a virtual grid could be used). This means that the layout might have to be examined in detail to allow for a more uniform division of the strip and to ensure that no components are split.

There does not appear to be any simple distributed algorithm to solve the constraint graph. Parallel algorithms do exist to solve the longest path problem but these algorithms do not seem to have an efficient implementation on loosely coupled processors. In any case the calculations of the longest path is not the bottle neck.

A distributed version of the constraint graph still shows promise since the graph generations phase is at best $O(n \log_2(n))$ and the longest path solution takes only $O(n)$. A speed up will be realized even if only the graph generation step is performed in parallel.

2.6.3 Shear Line

A distributed algorithm for layout compaction can also be derived from the shear line approach by subdividing the layout into strips. Each strip will be compacted individually and the results merged. Since the shear line technique only removes space, the layout has to be partitioned along shear lines. This requirement means that the entire layout must be examined in the partitioning process. As in the preceding approaches, the partitioning has to be performed in both compaction directions. The major disadvantage of the shear line approach is that its time complexity is $O(n^2)$, which is the largest time complexity of the three 1-D compaction techniques [3,6].

2.6.4 2-D Compaction

There are two ways to distribute the algorithms used by 2-D compaction approaches. One is to divide the layout into regions to be compacted separately, just like the 1-D compaction cases. The other method relies on the fact that certain 2-D compaction techniques use the branch-and-bound method to search for the optimal compaction. The search strategy used in branch and bound algorithms is easily distributed in most cases. The major problem with the 2-D approach is the exponential time required to solve the compaction problem in two dimensions [2].

2.7 Background on IPC and RPC

The proposed hardware configuration for the distributed compactor is a set of VAX's and SUN's connected via Ethernet and running 4.2/3BSD UNIX and version 3.2 of SUN's operating system (a 4.2/3BSD derivative). The Berkeley UNIX operating

system provides system primitives to perform interprocess communication. This section describes the UNIX Inter-Process Communication (IPC)[1][37,38] mechanism and SUN's implementations of Remote Procedure Calls (RPC) using this IPC mechanism. It is necessary to study the IPC system since it will determine the cost of sending information between the different processes of the distributed compactor.

2.7.1 Berkeley IPC

The basic idea of the Berkeley IPC mechanism is to make IPC similar to file input/output. In UNIX, file I/O is handled by a set of I/O descriptors, from which one reads and writes. Normally these descriptors are associated with a file on a disk but the IPC system associates these descriptors with communications paths between processes. By choosing to implement the IPC as part of the I/O system, the semantics of reading and writing have to be redefined. In normal file I/O, when a write operation is performed, the data is delivered to the I/O device and, when a read operation is performed, the data is transferred from the I/O device to the process. When the operations are performed in the context of interprocess communication these operations have different semantics. For example, a write operation does not **guarantee** delivery of the data to the receiving process nor does a read operation **guarantee** that data will eventually be received. Also, the associations of an I/O descriptor with the source or destination of data has to be defined.

The Berkeley UNIX interprocess communication is based on a descriptor called a socket. Like file descriptors, sockets are associated with the sending and receiving of data. For two processes to communicate, each of the two processes must have

[1]From a computer networks point of view the IPC facility is an example of a transport layer service interface.

a socket and the two sockets must be connected. Berkeley's IPC primitives create sockets, specify how these sockets are connected with other sockets, control how data is to be sent between sockets, perform the sending and receiving of data, and destruction of sockets. The style of communication and the particular protocol used by a socket is specified when the socket is created. **Streams** (i.e. virtual circuits) and **Datagrams** are the two styles of communication currently implemented by the Berkeley IPC system. The stream style of communication implies communication between two sockets only. This communication is reliable, error free, and does not persevere message boundaries. The UNIX concept of a **pipe** is implemented using the stream style of communication. The datagram style of communication implies that messages are individually addressed and are not guaranteed to arrive at their destinations. Datagram style sockets can allow communication to more than one socket at the price of unreliable delivery of the message. The two styles of communications lead to two types of sockets. Sockets using the stream style require an explicit connection before information can be exchanged. A socket created with the datagram style requires no connections and thus is a connectionless socket. The connection between these sockets and the manner in which the data is sent between sockets are important characteristics in the design of a distributed system based on this IPC system.

How connections are made between sockets will affect how the processes in a distributed system can be configured. In the IPC system there are two ways that sockets can be addressed, through the UNIX domain or the INTERNET domain. In the UNIX domain a socket is associated with a file name from a particular file system. A process can connect to another process only if it knows the UNIX file name of that process's socket. The UNIX domain is only available to processes

running on the same machine and therefore is not useful in a multi-processor distributed system. The INTERNET domain is an implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines and are treated as distinct messages which are delivered one at a time.

Depending on the domain, the different styles of communication are implemented with different protocols. A protocol is a set of rules, data formats and conventions that regulate the flow of data between participants in the communication. The performance for a style of communication depends on the protocol which implements that style. For the Internet domain (the domain used between machines) the protocol for the datagram style is called UDP (User Datagram Protocol) and for the stream style it is called TCP (Transmission Control Protocol).

Greater performance can be achieved by using the datagram style of communications instead of the stream style. The datagram style also allows messages to be broadcast from one socket to a set of sockets. In certain applications this broadcast ability is essential to realizing efficient distributed systems. For example, broadcast communications can be used to send common data to a set of processes. If the broadcast ability is not available, the information has to be retransmitted to all of the processes. The major disadvantage of datagram style communications is that the distributed system has to provide higher level protocols to guarantee reliable communication. The design and implementation of these protocols is not a trivial task.

Using the stream style of communication to send data between processes in a distributed system guarantees that all data sent will arrive. One disadvantage of

the stream style is that data can only be sent between two sockets (i.e. no broadcast ability). The protocol implementing the stream style can be prohibitive for certain applications.

2.7.2 SUN's RPC System

Sun's implementation of Remote Procedure Calls[39] follows the standard definition of a remote procedure in which a client process communicates with a server process via a procedure call mechanism. This is implemented by the client sending a message to the server process and the server process then invoking the specified procedure; when the calculations performed by the specified procedure finish, the results are sent back to the client process. Berkeley's IPC system provides the communication facility.

A major feature of this RPC system is the approach taken in transmitting the arguments and results of the procedure calls. Since the types of machines executing the server and client process can differ, the data representations of the arguments and results must be designed to cope with any incompatibilities caused by the differing architectures. An example of these incompatibilities are the byte orders of integers for VAXes and Motorola's 68000. These incompatibilities are taken care of by converting the data into a network standard called eXternal Data Representation (XDR)[40]. This conversion of the data to be output is called serializing, and the reverse process is called deserializing. Standard representations are defined for base types of characters, integers, and reals. Aggregate data types are defined in terms of the base types.

The steps involved in performing a remote procedure call are executed in the following sequence [41]:

1. The client calls the remote procedure by calling the procedure stub local to the client. This procedure serializes the arguments and sends a message containing the remote procedure to be executed and the serialized arguments to the server process.

2. The server process decodes the message and determines which procedure is to be executed. It then executes that procedure with the given arguments. When the procedure finishes, the results are serialized and sent back to the client process.

3. The client process receives the reply, deserializes the results, and passes the result to the procedure stub, which then returns form the procedure stub.

In the communication between the client and server processes, SUN's RPC allows the user to specify the communication style as either stream or datagram. This choice of communication style affects the performance and some of the semantics of the remote procedure calls. If the datagram style is used, some added abilities as well as restrictions are created for RPCs. Since the datagram style may be unreliable, the protocol used between the client and server has to be modified to ensure correct operation. SUN's current implementation also imposes a size limit on the arguments and results of a particular remote procedure call. The datagram style provides the ability to implement broadcast RPC. A broadcast RPC is defined as a procedure call to a set of server processes (i.e. a one-to-many). In a broadcast RPC, the client process has to accept the responses of all the responding server processes. A broadcast RPC provides an element of parallelism to the standard RPC mechanism. On some systems RPCs using the datagram style can be more efficient than RPCs based on the stream style.

When the stream style of communication is used for the RPCs there is no restriction on the size of the arguments and results. On the other hand, since the stream style guarantees data delivery the RPC system does not have to handle

communication problems. Since the stream style requires a connection between the communicating processes, broadcast RPCs cannot be implemented.

Another necessary feature of this RPC system is the name server function performed by a server called the portmap. In order for two processes to communicate, one of the processes must know the network address of the other process. Since remote procedure calls are only identified by their RPC number, before a client process can invoke a remote procedure it must obtain the network address of the server for that remote procedure. Once the network address is obtained the necessary connection can be made between the client and server processes. The portmap server translates from RPC numbers to network addresses. When a server process containing remote procedures is started, it sends a message to the portmap advertising its network address and indicating which RPCs it will handle. When a client makes a remote procedure call, the network address is first obtained from the portmap, so that the client and server can communicate. The portmap server is assigned a standard address so that all the clients and servers can communicate with it.

Chapter 3

Serial Compaction Algorithm

3.1 Introduction

The steps required to transform a symbolic layout description to a mask description will be covered in this chapter. Although the final goal of this thesis is to produce a distributed compaction algorithm, only the details concerned with the compaction process will be discussed at present. The design decision, of course, will be affected by the fact the final goal is to build a distributed system. The approach used in the design of this compactor is similar to those used in the MULGA and VIVID compactors, although it is closer to the MULGA approach in that it performs the spacing operation before the fleshing operation. The compaction discussed in the thesis assumes a flat symbolic layout (i.e. the layout contains no hierarchy). This translation process can be divided into the following steps: parsing the symbolic layout description, plotting the symbolic layout into a matrix form, netlist extraction, well formation, x-direction (horizontal) spacing calculation, y-direction (vertical) spacing calculation, and finally, generation of the mask description. The parsing

step converts the textual description of the layout to the internal data structures (i.e. a linked-list of records). In the compaction process all adjacent layout elements must be examined; the plotting step converts the layout description into a form which allows these comparisons to be efficiently performed. The netlist extraction step examines the layout so that an electric net can be assigned to all layout elements. The electric net is used in the spacing calculations. For CMOS technology, the layout primitives must be grouped into regions called wells. The well formation step is responsible for this grouping. After plotting, netlist extraction and well formation steps have been completed, the spacing of the virtual grid lines can be calculated. First, the spacing of the x-grid lines is performed, then the y-grid lines. Spacing calculations for the y-grid lines must also include corner checks. The last step consists of using the x-grid and y-grid line spacing and the symbol element table to produce the mask descriptions.

Netlist extraction, well generation, parsing, x and y spacing calculation steps all depend on the target processing technology. Since one of the main advantages of a symbolic layout system is the ability to generate mask descriptions for different and changing technologies, the design of any compactor system must be able to handle different manufacturing technologies. To make a compactor system technology independent, the steps in the compactor which are technology dependent must use some easily modifiable database system. The database must support the following pieces of information:

1. the types of the devices, wires, and contacts,
2. which of the primitives form electrical connections when they overlap,
3. what constraints exist in the formation of wells from primitives,
4. how the layout primitives are translated into mask features, and

5. what physical spacing must exist between the different mask features.

Items 3,4,5 are all derived from the design rules for a particular technology. For example, design rules which deal with the minimum separation of two layers would be contained in item 5. The last section in this chapter will discuss the technology database.

The following overall goals were established for the design of the compaction algorithm:

1. One of the main problems with compaction is the large amount of time required to compact layouts of moderate size. Therefore, the efficiency of the algorithms employed becomes of paramount importance.

2. The algorithms should work on local areas of the layout to allow partitioning of the problem into separate regions which could be assigned to separate processors.

3. Each step in the compaction process should also be independent from the other steps to the greatest possible extent, to aid in the partitioning.

4. The compactor should be technology independent and **all** technology specific details should be located in a database module. This provision should enable the compactor to be used easily with other technologies.

As with any set of goals, there are aspects of individual goals which have to be traded off, one against the other.

The remaining sections discuss each of the compactor's steps, the algorithms and data structures necessary to accomplish each step, and the justification for the chosen algorithms.

3.2 Parsing

The format chosen to represent the symbolic layout is a textual language similar to ICDL and ABCD. An example of a layout specified with ICDL appears in section 2.2.1. This format was chosen to provide compatibility with existing software which produces layouts in this form. The exact grammar for the language can be found in Appendix A. In order for the compactor to manipulate the description, it must read and parse the description to produce an internal data structure for the layout.

The LALR-1 parser produced by YACC [42] was chosen as the parsing algorithm for the system, mainly because of the ease of producing a parser with it.

The language is based on a subset of ABCD with extensions to handle well generation. The internal data structures must also represent these layout elements and their parameters. A listing of these primitives and their associated parameters is found in Table 3.1. The type parameter specifies the technology dependent parameter of each of the primitives. The name *metal2* is an example of a type for a wire in a specific technology. The parsing step must then know something about the particular technology, but because of the goal of isolating all the technology specific information in one module, an interface between that database technology and the parsing step must exist.

The parsing step passes the layout description to the plotting step. The layout description is represented by a list of layout primitives. Each item in the list will contain one of the three layout primitives and all the parameters for that primitive. In choosing the data structure to be used to represent the primitives list, two conflicting factors must be considered: 1) the number of primitives is not known

Device	Wire	Contact
position	position	position
type	type	type
width	width	path
len		
orientation		
net	net	net
well	well	well

Table 3.1: Primitives Parameters

until the entire description is parsed, and 2) the internal data structure has to be communicated to other processes in the network. The variability of the list size suggests the use of a dynamic list structure. If the parsing step is assigned to a separate processor, the dynamic structure can be difficult to transmit depending on the hardware configuration of the distributed system. If the textual layout description is scanned twice, then a dynamic list could be avoided at the expense of reading the description twice. The first pass would determine the size of the description and then allocate the necessary tables, which would then be filled during the second pass. This step could be omitted if the layout editor produced this sizing information. The question here is whether to perform two passes or to use a dynamic list which will have to be converted into a tabular form later on. For the system developed, a dynamic list was chosen to avoid the overhead of disk accesses.

3.3 Plotting

The plotting step is responsible for converting the symbolic layout description into a form which allows easy access to physically adjacent elements. Since the original symbolic description consists of a list of circuit elements, the plotting step is required

to organize the circuit elements according to their relative positions. This step then enables the spacing calculations between adjacent elements to be performed efficiently.

There are two immediately obvious ways to do this operation: 1) sort the list by each element's coordinate values and then scan the sorted list, or 2) plot a matrix where each entry corresponds to a grid point, and list all the elements which are located at that grid point. Both of these techniques have advantages and disadvantages. The sorting technique requires an $O(n \log(n))$ algorithm versus an $O(n)$ algorithm for the matrix plotting, where n is the number of elements. Sorting has the advantage of not requiring any more memory, while plotting requires allocation of a matrix the same size as the number of grid points contained in the bounding box of the layout. The matrix size is roughly linear with the number of circuit elements assuming reasonably dense layouts. If the sorting technique is used, the remaining steps of the compactor must use plane sweeping algorithms to examine the layout. Plane sweeping algorithms maintain a data structure of the relevant features at the sweeping line. In this case, at least two columns of the layout would contain the relevant features, some provision would have to be made for primitives which occupy more than one grid point. This data structure is required in order to deal with the devices and wires which occupy more than two grid points. The plotting technique was chosen because of its lower time complexity and apparent simplicity in comparison to the sorting method.[1] The MULGA and VIVID systems both use the plotting approach.

Figure 3.1 shows the basic algorithm for matrix plotting. Each element is ex-

[1]The memory requirements will become less and less of a concern with the newer workstations. The amount of memory used in a typical workstation is increasing to the 16-32 Mbyte level.

```
    Allocate matrix
    for each element in layout description
        case element.type
            contact:
x <- element.x
y <- element.y
                matrix[x,y] <- element
            device:
                plot_device(matrix, element)
            wire:
                plot_wire(matrix, element)
        endcase
    endfor
```

Figure 3.1: Psuedo-Code for Plotting Algorithm

amined and inserted into the matrix entries that it occupies. Special treatment is given to devices and wires since they occupy more than one grid point. A device occupies three grid points but its circuit description gives the coordinate of the gate and the devices orientation; therefore, the plot_device routine must calculate the grid points that the device's source and drain occupy. Since a wire can span many grid points, the plot_wire routine calculates each grid point spanned by the wire and plots it. The plot_wire routine is similar to a raster line drawing algorithm.

The matrix plotting method implies that the primary data structure is a matrix. There are several choices for the data structure for a matrix entry. A matrix entry must contain a list of all the layout elements whose coordinates correspond to that entry. Since a layout element can appear in more than one entry, it makes sense to have the entry contain a pointer to the layout element description and the type of element instead of the layout element description itself. Since the number of elements for a matrix entry is variable, a decision on whether to use a fixed size array or a dynamic list structure must be made. For simplicity, and since an upper

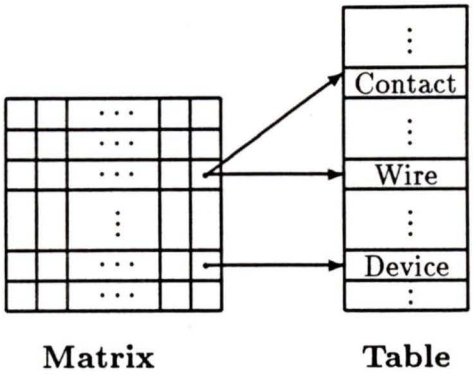


Figure 3.2: Matrix Data Structure

bound on the number of elements per entry can be determined, a fixed size array was chosen. The matrix plotting method then uses two data structures, the matrix of pointers to circuit elements and a table of the circuit elements. Figure 3.2 depicts this structure.

To expand a bit more on this step, a "work through" of an example layout for transmission gate is given. The textual description is shown in Figure 3.3, and the plot is found in Figure 3.4. This layout consists of two devices, four contacts, and six wires.

When this cell description is parsed, the bounding box is also calculated as ((8,11),(13,15)), where (8,11) is the lower left hand coordinate and (13,15) is the upper right hand coordinate. To plot each grid point, a matrix with height of (15-11)+1=5 and width of (13-8)+1=6 is allocated. Square brackets are used for matrix entries and parentheses for grid point coordinates. Assuming 0 is the starting coordinate for the matrix then the [0,0] entry corresponds to grid point (8,11), the leftmost point of the poly wire. For the first line describing a contact,

```
begin transmission
     contact    autocontact (9,14) or=n
     contact    autocontact (9,12) or=n
     contact    autocontact (11,14) or=n
     contact    autocontact (11,12) or=n
     wire       metal (9,14) (9,12)
     wire       metal (11,14) (11,12)
     wire       metal (11,13) (13,13)
     wire       metal (9,13) (8,13)
     wire       poly (10,12) (10,11) (8,11)
     wire       poly (8,15) (10,15) (10,14)
     device     n_type (10,12)
     device     p_type (10,14)
end transmission
```

Figure 3.3: Transmission Gate Description

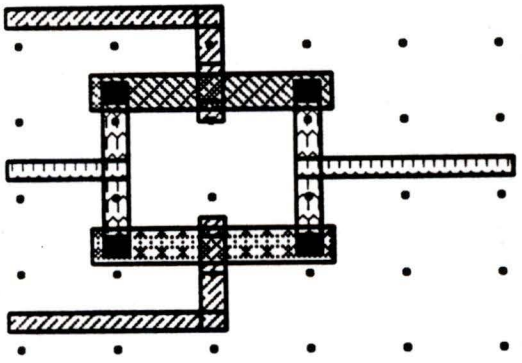


Figure 3.4: Transmission Gate Plot

```
contact    autocontact (9,14) or=n
```

it is located at point (9,14) and therefore an entry is made in the matrix at [9-8,14-11] =([1,3]). This entry contains a pointer to the contact's description in the layout elements table. The remaining contacts are plotted similarly.

Since transistors[2] span three grid points, the orientation and position of the gate are used to calculate the grid points for the drain and source of the device.

```
device     n_type (10,12)
```

For this case entries are made at locations [1,1],[2,1] and [3,1] corresponding to the grid points of the drain, gate and source of the device. An optimization can be made here by labeling the entries as drain, source, and gate instead of just device. If the entries were labeled just as device then the layout description entry would have to be accessed and the calculation repeated to determine which entry corresponded to the gate, drain, and source of the device.

The plotting of the wire,

```
wire       poly (10,12) (10,11) (8,11)
```

spans the grid points covered by the path. Entries are made of type wire which point to this wire's description. The matrix entries are plotted by starting at first point in the path and making entries along the line to the next point in the path. Entries are then made from that point to the next point. This process is repeated for all the points in the path. Thus for this example entries are made at [0,0], [1,0], [2,0], and [2,1].

[2]This discussion makes the assumption that all devices only occupy three grid points. This assumption is rather naive since bigger devices (i.e. channel width's and length's greater than 2) usually require more contacts than can be realized with only one grid point.

3.4 Netlist Extraction

The netlist of a symbolic layout describes the circuit represented by the layout. During the compaction process the spacing of layout elements is affected by the electric net of the elements. For example, if two poly wires are of the same electric node and are adjacent then the spacing rules between the two wires can be ignored. For a symbolic layout the netlist can be obtained easily and efficiently.

Each of the three types of elements in a symbolic layout has from one to three nets associated with it. A device has three nets, one each for the drain, source and gate. A wire has only one net associated with it. A contact, also, has only one net associated with it. The net extraction step has to determine which elements share common nets. One way to perform this step is to initially assign unique numbers (a number represents a net) to each element in the layout. Wires and contacts would be assigned one number each, and a device would be assigned three. This initial assignment assumes that all of the components are not connected. During the netlist generation process as the components are determined to be electrically connected the net numbers of the components can be grouped into sets to denote the fact that these components are connected. This means that each net is represented as a set of net numbers. After all of the elements have been examined the netlist will consist of a list of sets of numbers. Two elements are then electrically connected if their assigned numbers are in the same set. The technology database has to be consulted to determine if overlapping elements are connected.

The set building operation can be performed in conjunction with the plotting step. As each element is plotted, a check is made to see whether the elements present share the same electrically conductive layers and, if so, the nets for the elements are

merged. The building of these sets can be performed very efficiently if the union of disjoint sets construct [43, page 70] is used. In this case, the time complexity of the net extraction process is the same as the plotting step. To simplify the specification of the netlist all layout element numbers which are members of the same set will be changed to some unique number. Two layout elements will be electrically connected if their node numbers are equal.

One issue posed by the construction of the netlist is whether it should be performed by the compaction process at all. The layout language could be expanded so that each element has as a parameter the net to which it is connected. The system which generated the layout could easily keep track of the netlist as the layout was constructed. If the layout generation system had this capability, then a netlist extraction step would not be required. Another issue is that the netlist is global to the entire layout but if the layout is partitioned for the compaction algorithm then the netlists for each partition must be merged to give the complete netlist. This issue can be ignored if the connectivity information is not used in the spacing calculations when the layout partitions are joined back together. Since this issue deals with the distribution of the compaction algorithm it will be discussed again in section 4.3.3.

To facilitate use with other systems, the implementation of this compactor can extract the netlist if one is not provided.

3.5 Well Generation

In the generation of mask layout for CMOS technologies the n-type and p-type devices have to be placed in different substrates. The regions which define these

substrates are referred to as wells. The assignment of devices to wells can be done either by the layout designer or by the compactor system. In a decision between these two approaches there is no clear winner. The major problem with getting the compactor system to perform the groupings of layout primitives is that the required algorithms have expensive time complexities. The construction of wells is also a technology-dependent problem because some technologies have restrictions on well formation.

The approach taken for the current implementation is to have the layout designer specify the well placement. The layout language is modified to include a well parameter for each primitive. For example,

```
device    n_type (10,12)  well=47
```

associates well number 47 with the n_type device located at (10,12).

3.6 Horizontal Spacing

Calculation of the grid spacing is the essential feature of any virtual grid compactor. This is the step which determines how "good" the compactor is, where the "goodness" is measured by the area of the mask layout. This is also the step which tries to capture the complicated design rules as a simple set of spacing calculations.

Horizontal spacing consists of finding the distance between vertical grid lines. Since the spacing of the horizontal grid lines has not yet been calculated, the distance between the vertical grid lines is determined solely for the required separation of elements on the same horizontal grid line. Section 2.2 shows how the matrix entries are examined to determine the vertical grid line spacing. Just as in the VIVID system a frontier data structure is maintained.

The basic operation in the horizontal spacing calculation is to determine the spacing between two adjacent grid points on the same horizontal grid line. This is accomplished by generating the mask layers present at each grid point, combining the mask layers of the leftmost column with the frontier and then using the design rules to determine the spacing that should exist between the two grid points. The spacing is affected by the electric net of the layers, the position of well boundaries and the types of the layers. When the spacing calculation for each column is completed, the frontier is merged with the column's outlines.

There are several issues associated with the design of the spacing calculation module:

1. how much, if any, of the knowledge of the layout primitive should be used,
2. are well layers treated the same as non-well layers, and
3. how should devices be handled?

The spacing calculation can be optimized by examining which layout primitives are present at the two grid points. For example, if the grid points being compared contain entries pointing to the same wire then those entries can be ignored. To use this knowledge a complicated set of rules detailing how the different layout primitives interact would have to be constructed for each technology. Another advantage of using this level of knowledge is that devices can be treated as a unit, instead of having to consider a device as a set of separate primitives.

The distinction between well layers and non-well layers is that non-well layers are associated with a particular layout primitive, while well layers are associated with a set of primitives (region of the layout). This poses the problem of how the well layers are generated, either by the primitives or by a separate well generating

step. If well layers are generated by the primitives, then, unlike the non-well layers which can be merged if the layers are electrically connected, some other mechanism must be used which allows well layers from the same well to be merged.

Another problem stemming from the way grid spacing is calculated is that the device layout primitive occupies three grid points and must be fragmented into three distinct entries. This means that the formation of the mask layout for a device is broken up into several disjoint steps. This fragmentation forces an unnatural generation of the device mask representation. The difficulties of this approach requiring the fragmentations of the primitives can be seen in the next section.

Taking the above issues into consideration, the design of the current spacing calculation is presented in the next section.

3.7 Spacing Between Two Grid Points

There are two steps used in calculating the spacing between grid points: 1) determine what mask features are present, and 2) use the manufacturer's design rules to determine the minimum separation between the mask features at the two grid points. Since the compactor has to work with different technologies, the mask features must be represented abstractly. Mask features are represented as polygons with attributes of type, well number, and electric net number. Figure 3.5a shows a symbolic layout and Figure 3.5b shows the mask layers generated by each of the primitives. In Figure 3.5b the spacing between grid lines is not yet determined. Examining the mask layers at grid point (3,4) and (4,4), it can be seen that only the features to the right of (3,4) and to the left of (4,4) interact. This observation suggests that for the spacing calculations only the outlines of the mask features

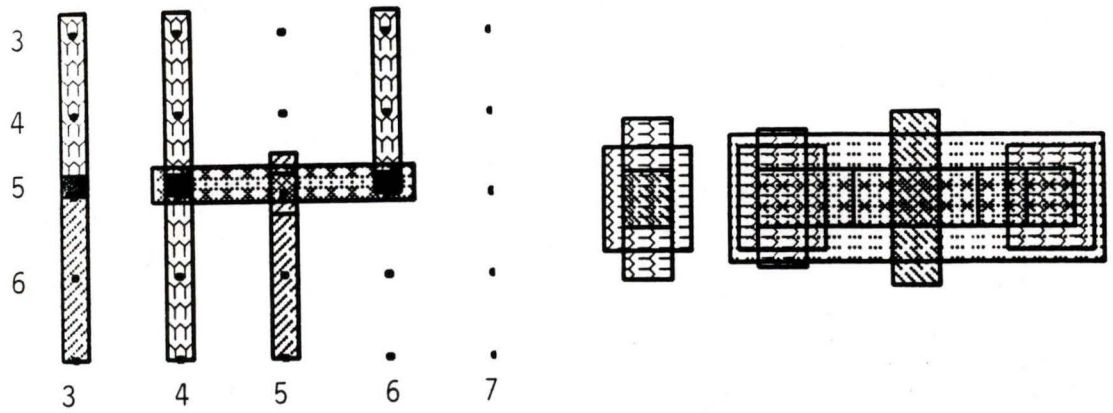


Figure 3.5: Cell Compaction

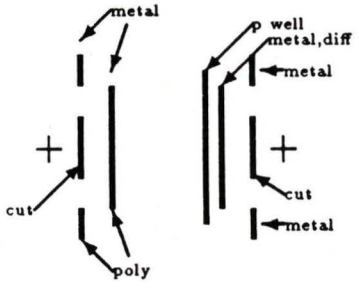


Figure 3.6: Outlines for Spacing Calculations

facing each other are relevant. Figure 3.6 shows the outlines of the mask features for the grid points (3,4) and (4,4).

An outline is specified as a line segment of a given size placed at a given distance away from the grid point. The line segment is centered around the grid line connecting the two grid points. To perform the spacing calculation the compactor first determines the outlines for the two grid points. The technology database must be consulted to determine what layers are present for each of the layout primitives.

An outline is then constructed for each mask layer. In the compaction process, if more than one polygon of the same layer is present, the outline of the polygon with the greatest extent is chosen as the base for the outline. The outline for the leftmost grid point (i.e. the direction of compaction) is merged with the outline of the frontier.

The spacing calculations between the two outlines are performed in three phases. First the spacing between all non-well layers of the same type is calculated. For this phase, if the electric nets of the two outlines match then the separation is set to 0. Otherwise the spacing is set to the minimum separation distance for that layer summed with the lengths of the outlines. A record of any ignored spacing constraint must be made to enable the filling in of anti-features. An anti-feature is the space left between two polygons of the same mask layer which are placed closer together than the minimum allowed design rule. The next phase checks the spacing for all non-well layers against each other, with the exception of layers of the same type. Problems occur in setting the spacing between the gate region of a device and the source or drain regions because these regions occupy separate grid points. For MOS technology, the gate region is made by overlapping poly and diffusion layers, while the drain or source region is composed of a diffusion layer. A spacing calculation between the poly of the gate and the diffusion of the drain or source is performed because the drain and source are on different grid points from the gate. Unfortunately, this calculation is erroneous since the diffusion layer under the gate poly is the same as in the drain and source regions. This complication illustrates the difficulty of designing a simple spacing calculation to handle complex design rules, and the problems of treating a single device as three separate parts. The way the compactor handles this case is by testing if the layers present at the two grid points

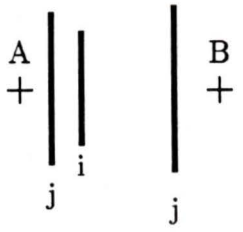


Figure 3.7: Merging of Grid Points

are mergable. Figure 3.7 shows the outline of cases where layers are mergable. The spacing calculation between layer i at grid point A and layer j at grid point B can be ignored if the following conditions are true (thus layer j is mergable):

1. layer j is also present at grid point A,
2. the electric net of the j layer are the same (this implies that the j layer spans from A to B) and,
3. the size of layer j at A is greater than or equal to the size of layer j at B.

If these conditions were true then the j layers are mergable and the spacing calculations for layer i and layer j can be skipped.

The final phase of the spacing calculation for two grid points is to determine the spacing between well layers themselves and between well layers and non-well layers. This approach assumes that the well layers are generated for each grid point. The spacing calculation is qualified by the well to which the layers belong. No calculation is performed if the well layers are part of the same well.

The spacing between the grid points is the maximum of the individual spacing calculations. This can be expressed by the following equation:

$$\text{space} = \max(\text{dist}_i + \text{dist}_j + \text{DR}(i,j) \quad) \quad \forall \quad i,j$$

where i,j are layers, $dist_i$ is the distance from the grid point to the outline of i, and $DR(i,j)$ is the minimum separation of layer i and layer j.

3.8 Vertical Spacing

The vertical spacing step calculates the separation of the horizontal grid lines (y-grid) so that no design rule violations occur. This calculation is performed between two y-grid lines at a time until all adjacent grid lines have been compared. The comparison of two y-grid lines is complicated by the spacing of the x-grid lines being already set. Unlike the x-grid lines where only the adjacent grid points of the two lines had to be examined, all of the grid points for the two lines within the maximum corner design rule distance have to be compared. Figure 3.8 shows which grid points should be compared for horizontal spacing and which grid points should be compared for vertical spacing. Since the spacing of the grid points on the x-grid lines has been set by the horizontal spacing step, the mask layers present at nonadjacent grid points can violate design rules, when the spacing between y-grid lines is calculated; thus, the spacing of these grid points must also be determined.

The procedure for calculating the spacing of two grid points is similar to the one used for the horizontal spacing except that all distance calculations have to include both vertical and horizontal distances. Figure 3.9 shows the outlines of the mask layers at each grid point for two y-grid lines and how the distance between the outlines are measured. The Euclidean distance is used as the distance between outlines unless the projection on the y axis of the two outlines overlap; only the horizontal distance is then used. The spacing of the grid points for horizontal compaction can use the same procedure as the vertical compaction except that the

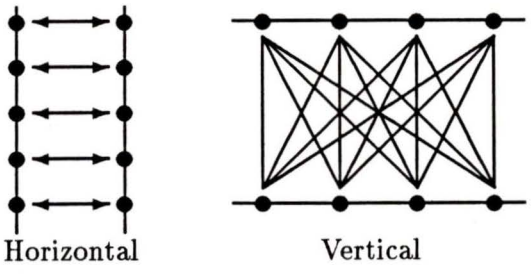


Figure 3.8: Horizontal and Vertical Spacing Checks

vertical distance would always be zero. The types of measurements are exactly the same as the ones performed in the horizontal spacing.

The checking of the nonadjacent grid points ensures that none of the corner design rules are violated. The equation for the spacing calculation for vertical compaction is (see Figure 3.9):

$$\text{space} = \max(\sqrt{(\text{DR}(i,j)^2 - (\text{xgrid} - (size_i + size_j)/2)^2)}) \quad \forall \quad i,j$$

3.9 Fleshing

The fleshing step of the compactor uses the x and y grid spacing produced by the spacing calculations and symbolic primitive list to generate the mask layout. The fleshing step also fills in the anti-features and merges the mask well layers of the symbolic primitives which belong to the same well. Since the type and formation of the mask layers depend on the manufacturing technology the fleshing step uses the compactor's technology database to access this information.

A mapping from virtual grid coordinates to physical mask coordinates is con-

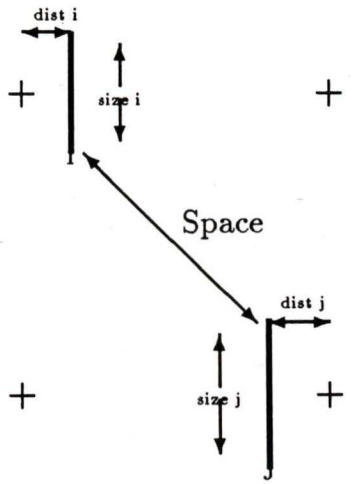


Figure 3.9: Corner Rule Calculation

structed from the x and y grid spacings. The mapping for the x coordinate is constructed by assigning to each vertical grid line the sum of all the spacings from the leftmost vertical line to itself. The construction of the mapping for the y coordinates is similar except, that the horizontal grid lines are used. This mapping, together with the description of a symbolic primitives is passed to the technology database to produce the mask layers for each primitive. The mask layers from each symbolic primitive are then combined to produce the final mask layout.

3.10 Technology Database

The output of a compactor is a mask description used to manufacture an IC. A mask description consists of a set of mask layers, where a mask layer is composed of polygons. The types of mask layers and constraints on the placement and size of the polygons are specified by the IC manufacturing process. Although the symbolic description only partially specifies some of the mask layers, the technology database must specify all the mask layers for a particular technology. The mask layers are the

device	n_type	p_type				
contact	mpd	mnd	mp	via	vss	vdd
wire	poly	metal	metal2	ndiff	pdiff	

Table 3.2: CMOS technology types

fundamental data objects in the database. All other information in the technology database refers to the mask layers. Each technology database therefore has the mask layers defined as a starting point.

For virtual grid compaction the technology-specific details can be divided into four areas: types of layout primitives, primitives which are connected when overlapped, translation of primitives to mask representation, and spacing between mask polygons. The information in each of these areas is used by certain steps in the compaction process. Information about the types of the layout primitives is used in the parsing, netlist generation, spacing, and fleshing steps. The connectivity information is used in the netlist generation step. The translation to mask representation data is used in the spacing and fleshing steps. The layer spacing data is of course used in the spacing steps.

3.10.1 Layout Primitive Types

For each manufacturing technology the types of the contacts, wires and devices may be different. Table 3.2 gives an example of the types for a particular CMOS technology.

Since these types are parameters to the contact, wire, and device description in the layout language, the parsing step must know which types are valid for the current technology. The technology module will have to maintain the types for

a technology and support an interface to the parsing step which will allow the parsing step to validate the type parameter of the layout primitives. These names can be represented by simple tables containing the type names. The parsing step translates these names into an internal representation which is then used by the netlist generation, spacing, and fleshing steps.

3.10.2 Netlist Connectivity

The netlist connectivity part of the database determines which layout primitives are electrically connected when they occupy the same grid point. This is implemented, in constant time, by a simple table lookup indexed by the type of the two layout primitives. The connectivity of the primitives is determined by the mask layers present for each primitive. By definition, contacts have 1 net, wires have 1 net, and devices have 3 nets associated with them.

3.10.3 Mask Generation

The mask generation database translates the symbolic layout primitives to the technology dependent geometric primitives. These geometric primitives, typically isorectangular, are grouped into mask layers. The generation of the mask primitives is performed in accordance with the non-spacing design rules for a particular manufacturing process. For example, the geometric primitives generated for a wire have to be as large as the minimum feature sizes of the layers which are used in that wire. Although the spacing and fleshing steps both use the mask generation database, they require different representations of the same information. The spacing steps need to know what geometric primitives are present at each grid point, while the fleshing steps generate all the geometry for each symbolic primitive in one opera-

tion. For the fleshing step, the symbolic primitive and absolute physical coordinates of the symbol's grid points are required as inputs to the database to produce the mask layout. The remaining discussion of the mask generation database will consider the details of the database for the fleshing and spacing steps separately. Even though the fleshing and spacing information is accessed differently, it is still the same information.

The spacing step determines the mask layers present at a grid point by passing to the database the symbolic primitive or the part of the primitive at that grid point. For example, the mask layers for the source of a device are requested and not the layers for the entire device. The format chosen to represent the geometric primitives is governed by the requirements of the spacing step. A data structure representing the width and height of a rectangle was chosen as the format for the geometric primitives. Each rectangle also has the attributes of the mask layer, electric net, and well number. These attributes are directly derived from the description of the symbolic primitive. The geometric primitives for a particular grid point are returned as a list of rectangles.

The fleshing steps output the geometric primitives in the format used by the mask layout tool CAESAR [44]. Any of the other mask layout languages like CIF [45], or EDIF [28] could have been just as easily used.

A simple table lookup implementation for the mask generation is not feasible, since the mask generated for each primitive depends on the parameters of the symbolic primitives. This suggest that the database contains a procedural description of how each primitive is translated into the mask data. The layout language supports the three primitives – wires, contacts, and devices. A discussion of the procedures for each of the symbolic primitives used to generate the geometry for the spacing

step will be given, followed by the procedures for the fleshing step.

In generating the mask layers for different primitives, the number of grid points occupied by a primitive will determine the complexity of the generation process. Since the contact primitive only occupies one grid point the generation process depends only on the parameters of the contact. Since devices occupy three grid points, there must be procedures for each of the grid points. Also, the masks produced for each grid point must combine to correctly form the mask layout of the device. A problem with generating the geometric primitives for wires is that the physical size of the wire is only defined for the width of the wire. The other direction, the direction of the wire path, is determined by the spacing of the x or y grid lines. The undefined size for the geometry of a wire's grid point is set to be the same as the width required to handle the case where the wire changes direction (i.e. a bend in the wire). This fixed size will not affect the final spacing because the geometry of a wire will share the same electric net and thus the spacing calculation between grid points with masks layers from the same wire will not be performed.

The generation for the geometric primitives in the fleshing step is much simpler because the spacing of the x and y grid lines is fixed, and since the geometry for each primitive is produced all together.

Although the layout masks generated by the fleshing steps and spacing calculation produce the same layout, mask generation is easy in the fleshing step since the x and y grid spacing have been fixed. The same procedure can be used for mask generation for contacts since contacts only occupy one grid point. For a device the three physical coordinates are used with the parameters to generate the layout. Geometric primitives are generated for each segment of a wire and not for each grid point. The only information required to generate the mask layout for wire segments

is the segments ends and the wire's width.

3.10.4 Spacing Rules

The spacing rules are organized as set of two dimensional matrices indexed, for both indices, by the mask layer. Each matrix in the set contains the spacing entries for a particular context (i.e. inside or outside of a well). The entries in each matrix contain the minimum spacing. Entry [i,j] of the matrix contains the minimum spacing between layer i and layer j.

Chapter 4

Distributed Compaction Algorithm

4.1 Introduction

This chapter shows how the compaction algorithm described in Chapter 3 was adapted to run on a distributed system. The goal of this adaptation is to realize a speed up of the compaction algorithm which is linear in the number of processors used. In the design of a distributed compactor two factors are considered: 1) the hardware and type of runtime system, and 2) the parts of the compaction problem that can be solved in parallel.

The hardware of the distributed system used in this thesis consists of a set of fast processors connected together by a local area network, such as Ethernet. This is a typical configuration found in industry and universities. This hardware organization implies that a distributed version of the compactor will be divided into a group of processes running on separate processors interchanging information via

the local area network. How the compaction algorithm is partitioned will depend on the relative speed of the processors and the cost of communication.

The compaction problem can be solved on other types of distributed or parallel computing systems. The resulting partitioning or algorithm design is strongly affected by the type of system. The two most common types of distributed systems are those based on the SIMD (Single Instruction Multiple Data stream) model or the MIMD (Multiple Instructions Multiple Data stream) model. In a SIMD machine implementation, an efficient compaction algorithm partitioning would be one where the data was organized into arrays where uniform operation could be performed. The greater the organization the greater the realized speed up will be. For systems which follow the MIMD model the connection between the processors will affect the design. If the processors in a distributed system are more closely coupled, then finer levels of parallelism are possible. As the communication cost increases the problem has to be divided into more disjoint subproblems to realize any speed enhancement. One of the finest level of parallelism is where the information about a single grid point is stored in a single processor. As the cost of communication increases, so does the requirement to reduce communication by clustering the smaller problems.

The steps of the serial compaction algorithm provide an initial partitioning for the distributed version of the compactor. These steps are good candidates for the initial partitioning of the compactor since these steps already partition the compactor into areas which perform well defined operations on the input to the compactor. These steps must be examined for parallelism, and for the amount of data produced by each step and required by each step. The parallelism of a step will determine the speed up which can be achieved by distributing the step to multiple processors. The amount of data produced and required will determine

the communication cost of distributing the step across the network. Moreover, the inter-dependence of the compaction steps has to be determined.

The procedure used to assign the compactor's steps to processes is discussed in section 4.2. In section 4.3, the analysis given in item 1 of the procedure outlined in section 4.2 is applied to each of the compactor's steps. The subsections in section 4.3, will show how application of items 2, 3 and 4 to the compactor's steps will produce a distributed system following the client/server model. The client/server model consists of one process, the **client**, which requests service from **server** processes. The server processes perform work in parallel if the servers are located at different processors.

The implementation of the compactor using SUN's Remote Procedure Calls (RPCs) will be given in the final section of this chapter, with attention focused on how the RPC mechanism was modified to handle parallel execution.

4.2 Assignment Procedure

For a distributed system based on an MIMD architecture the amount of improvement in the runtime performance of an algorithm is directly related to the assignment of parts of the algorithm to the individual processors. In the development of the distributed compactor the following procedure was used to assign the serial steps of the compactor to processes/processors.

1. For each step in the algorithm determine:
   (a) the amount of data required as input,
   (b) the amount of data produced as output,
   (c) which steps must precede it, and
   (d) the amount of parallelism that exists for this step

Items 2, 3, 4 of the procedure are repeated for each process.

2. For the steps which have not been examined, select a step which does not require data directly or indirectly from the other unexamined steps.

3. For the selected step, determine which previously selected steps provide input to this step. At this point a decision has to be made as to whether to include this step in the same process(es) as the steps which provide it with data or to place it in a separate process. This decision is based on two factors, the degree of parallelism of the selected step and the communication cost associated with the assignment of the step to a new process. If the selected step contains no parallelism, then the step should be included in the same process as the one supplying data to it in order to eliminate communication costs.

4. From the set of processes already created, see if any can be merged, thus simplifying the system, without reducing the parallelism.

In some cases it might be very difficult to evaluate certain assignments. For these cases the different alternatives should be tried and the one with the best performance chosen.

4.3 Analysis of the Compactor's Steps

To obtain a decrease in the run time of any algorithm, the algorithm must include steps which can be performed in parallel. If all the steps of a particular algorithm require a strict sequential order of execution then that algorithm could not be adapted to run more quickly even if more processors were available. Even if a particular step of the compactor can be executed in parallel, the communication cost of transmitting the input and output of that step could take as much time as is saved by the parallel execution. Each of the steps must be examined to see if that step can be transformed into one which can execute in parallel and if the parallel execution will not be offset by the communication time.

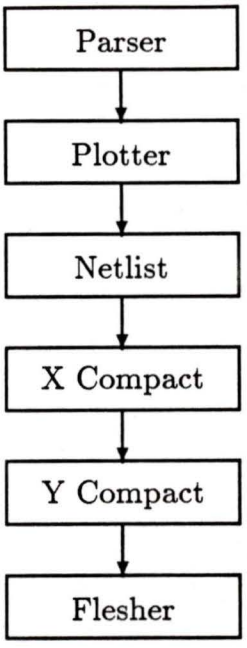


Figure 4.1: Compactor's Steps

Since each of the steps of the compactor relies on the technology database, the relevant database information must be distributed along with each step which is assigned to a separate processor. In the single processor version of the compactor, the steps are performed in the following sequence: parsing, plotting, net generation, horizontal spacing, vertical spacing, and fleshing (see Figure 4.1). In the distributed version of the compactor these steps must still occur in the same sequence.

4.3.1 Parsing

The first step of the compactor consists of parsing the symbolic layout and producing an internal representation of it. This step is sequential in nature since the input

symbolic layout description is a textual language which must be parsed serially. Also the information is physically located on a disk local to one processor. The input for this step is the textual symbolic description and the output is the internal description of the symbolic layout. This internal description is composed of two arrays, one array for the symbolic layout elements and another array containing the points in the paths of wires. The textual description is transformed into this format to provide random access to symbolic layout elements. The internal description is passed to the plotting step. Obviously, this step has to be performed first and none of the other compactor steps can be started in parallel with it. If the layout primitives were ordered in some sequence in the description then some of the other compactor steps could be started in a pipeline fashion. Table 4.1 gives typical data input and output sizes. The cell layouts used to test and evaluate the compactor are CMOS cells ranging in size and complexity from simple NAND gates to a memory array. Section 5.2 describes each of the cells in more detail.

4.3.2 Plotting

The plotting step (i.e. converting the symbolic description to a matrix form), can be performed in parallel since the symbolic layout can be partitioned into separate regions and each region can be plotted independently. Two possible ways of performing this partitioning are: 1) to separate the symbolic description and then send sections of the description to the individual processors, or, 2) to send the entire description to the separate processors and then have each process scan the list and only plot the relevant layout elements for that process. The first method requires an $O(n)$ partitioning step which then reduces the communication time and the time taken to plot the partitioned regions. The communication time is reduced since

cell name	input size (bytes)	num. of primitives	output size (bytes)
nand_4	2059	33	1188
nand_8	3862	62	2232
ff	18207	355	12780
tt2	18709	396	14256
tt3	30751	645	23220
tt4	43904	919	33084
tt5	59038	1233	44388
tt6	76891	1603	57708
tt10	165278	3428	123408
tt20	555109	11158	401688
shift_30	543232	8550	307800
bit	1491251	23275	837900
mem	1979368	31768	1143648

Table 4.1: Typical Parser Input

only some sections of the symbolic description are sent to each processor. Also memory usage and plotting time are slightly reduced, since only the relevant layout primitives are present.

The second method does not require the partitioning of the symbolic description, but the entire symbolic list has to be communicated to each processor and slightly more time is spent in the plotting step, since the irrelevant elements still have to be examined. This approach also means that each server can only manipulate the region transmitted to it. Each region the server works on has to be sent individually. If the distributed system can broadcast to all processors then the communication time for both methods should be equivalent within a constant factor. The choice of method depends on the type and cost of the inter-processor communication. The partitioning of the symbolic description must occur before the plotting step and it can only be done by one processor since the description is located at only one

cell name	input size (bytes)	matrix height	matrix width	matrix size (bytes)
nand_4	1188	7	12	3528
nand_8	2232	9	20	7560
ff	12780	23	37	35742
tt2	14256	31	27	35154
tt3	23220	55	35	80850
tt4	33084	61	44	112728
tt5	44388	67	53	149142
tt6	57708	73	62	190092
tt10	123408	79	99	328482
tt20	401688	139	189	1103382
shift_30	307800	106	181	805812
bit	837900	176	295	2180640
mem	1143648	282	237	2807028

Table 4.2: Plotting Step Inputs

processor. The input of the plotting step is the symbolic description and the output is a matrix for the layout of the elements.

The size of the matrix plotted is approximately linear with the number of symbolic circuit elements; typical sizes of input descriptions and output plots are given in Table 4.2. More detail description of the layout cells is found in section 5.2.

4.3.3 Net Generation

The algorithm used for net generation is closely tied with the plotting step and thus can be distributed in the same manner. To reduce communication, the net generation step can be assigned to the same processor as the plotting step. This inclusion of the net generation step with the plotting step unfortunately results in only a partial netlist available to each processor since each process examines only part of the layout. The resulting nets must be merged to produce the netlist of the

entire layout. The other option is not to merge the netlist. The incomplete netlist will affect some of the spacing calculations; however, this effect should be small.

The merging of the netlist can be performed in two ways. Either each processor performing netlist extraction communicates its partial netlist to a merging process, or the processes communicate with each other and each process performs the merging operations by itself. This step can be skipped if the symbolic description already contains the netlist information. The input for this step is the symbolic layout in matrix form and the output is the netlist for the layout. The net generation step produces as output the electric net number for each of the input layout primitives. Since a maximum of three net numbers can be produced for each primitive, the maximum output generated by this step is bounded by three times the number of layout elements in the plotted matrix. The number of nets is also bounded by the number of virtual grid points.

4.3.4 Horizontal Spacing

The observation that the horizontal spacing between layout elements only depends on adjacent elements implies that the horizontal spacing calculation can be performed in parallel for each local region. In the serial case, horizontal spacing is performed by scanning the plotted layout in the direction of compaction, calculating the spacing between pairs of grid points in the vertical columns, constructing the frontier, and finding the pair of grid points with the maximum spacing for each pair of vertical columns. The spacing of a pair of horizontal grid points depends only on which symbolic elements are present at the two grid points and the frontier of grid points adjacent to the two grid points. This means that the spacing calculation for horizontal compaction can occur in parallel to the same degree that the plotted

matrix is partitioned. Maximum parallelism occurs when there is a processor for every two grid points. In practice this degree of parallelism cannot be obtained since the spacing of the two grid points depends on the environment (frontier) to the left of the grid points. This means that for the final spacing to be calculated the grid points to the left also have to be accessed. Fortunately this effect, in most cases, is restricted to only 3 or 4 grid lines to the left, so that the spacing locality assumption is not violated. The locality assumptions for layout compaction is that the spacing for any two grid lines is only dependent on the immediate neighbourhood. The spacing for any two vertical grid lines depends on the spacing of each grid point pair along those grid lines. Since the spacing of the grid lines is the maximum of the spacing of all the grid point pairs, if the matrix is partitioned along the vertical columns all the pairs of spacings will be located in the same process and then can be easily combined. As the number of available processors decreases, then the width of these vertical partitions can be increased to share the amount of work among the processor set. By partitioning the matrix in this manner the communication required between the partitions in order to complete the spacing calculation is minimized.

To produce the horizontal spacing for the complete layout the spacing of the local regions must be combined. The output of this step is an array of spacings between the vertical grid lines. The size of the array depends on the width of the plotted matrix and thus on the number of vertical grid lines in the layout to be compacted. For N vertical grid lines there are N-1 entries in the spacing array.

4.3.5 Vertical Spacing

The assumption that spatially separate layout primitives do not affect the spacing of each other is still valid for the vertical spacing calculation. This locality principle means that the layout can be partitioned and the spacing calculation for the partitions can be performed in parallel. Unlike horizontal spacing where pairs of grid points can be examined in isolation, in vertical spacing the set of grid points with possible corner rule interaction must be examined. This means that the smallest region for which vertical spacing can be performed independently consists of sections of two rows of the layout.

Since the horizontal spacing results are necessary to calculate any corner rule spacing, the vertical spacing calculation must follow the horizontal spacing calculation. The output of the vertical spacing step is an array of the spacing values for every pair of horizontal grid lines. The height of the layout to be compacted minus 1 will give the number of elements in the vertical spacing array.

4.3.6 Fleshing

The fleshing step, the final step, cannot begin until the horizontal and vertical spacing steps have produced the x and y grid spacing. The x and y grid spacings are used to calculate the mapping from virtual grid coordinates to physical mask coordinates. If the symbolic layout is partitioned into regions then the fleshing step can be performed in parallel for each of these regions. Since the output of this step must eventually be stored on disk, the outputs of the separate fleshing processes have to be combined. The decision on distributing the fleshing step depends on the amount of output produced and the communication of this output to the processor which contains the disk. Table 4.3 gives typical values in bytes of the amount of

data produced by this step.

cell name	output (bytes)
nand_4	2032
nand_8	4102
ff	21219
tt2	18616
tt3	29953
tt4	42679
tt5	57609
tt6	75341
tt10	158623
tt20	522715
shift_30	593322
bit	1654550
mem	1962106

Table 4.3: Fleshing Output

4.4 Assignments for the Compaction Algorithm

A possible assignment of steps to processes is shown in Figure 4.1. Each step shown in a separate box is treated as a process. Unfortunately this assignment does not consider the communication cost. To produce an efficient distributed compactor the procedure discussed in section 4.2 is used.

The first step chosen is the parsing step since all remaining steps of the compactor depend on the symbolic layout description. The next step is the plotting step because it transforms the symbolic description into a matrix used by the remaining steps. Since the analysis section shows that this step could be performed in parallel, the plotter step is assigned to a set of processes. Each plotting process

is assigned a region of the layout to be plotted. To provide uniform work for each process, the size of each region should be roughly equal. The system being developed is composed of a process which parses the symbolic descriptions and a set of processes which plot regions of the layout. The parsing process assigns the region of the layout for which each of the plotter processes is responsible. The symbolic description of the region with which the plotting process has to work is also sent to each of the plotting processes by the parsing process.

The net generation step is the next step to be selected according to the compactor's step dependence. The algorithm chosen to implement the net generation step requires that the symbolic layout be plotted. If this step is included with the processes containing the plotting step then the nets could be generated for the regions assigned to the individual plotting processes. This assignment produces a netlist for each region which would have to be merged to produce the netlist for the entire layout. The merging would require communication of partial netlist information among the netlist processes. Assigning the net generation step to separate processes would require communication of the plotted matrix from the plotting process to the net generation processes. From the discussion in section 4.3.3 in can be shown that no extra parallelism can be realized by separate assignment of the plotting and net generation steps, and therefore the net generation step is assigned to the same processes as the plotting step. The nets of separate strips are not merged to save on communication time.

From the remaining steps which have not been selected the horizontal spacing step input has been produced by the already selected steps, so it can be chosen next. The spacing calculations for this step can be performed in parallel for separate regions of the layout. Since the spacing step uses the matrix and netlist produced

by the plotting step in its calculation, the output of each of the plotting processes is sent directly to the separate horizontal spacing processes. The spacing calculation from the separate horizontal spacing processes would have to be combined before the vertical spacing step could start. The way in which the layout is partitioned into separate regions for the plotting steps affects the efficiency of the parallel execution of the horizontal spacing. By partitioning the layout so that the grid points in the same vertical grid line are in the same local regions, the amount of information exchanged to merge the spacing calculation is reduced. This means that for efficient horizontal spacing calculation the layout should be partitioned into vertical strips. This efficiency refers to how much information has to be exchanged between separate processes. Since each of the plotting processes is paired with a spacing process and the spacing step must follow the plotting step, the spacing step can be assigned to the processes containing the paired plotting step.

A process must be created to merge the results of each of the horizontal spacing processes so that the horizontal grid (x grid) can be calculated. The complexity of the merging process can be reduced by carefully partitioning the layout into the regions assigned to the plotting-horizontal spacing processes. If the layout is divided into uniform strips, then in order to merge the adjacent strips the grid points and frontiers of the adjacent grid lines must be compared. If layout is partitioned so that the strips overlap by a few grid lines, then, at the expense of performing some redundant calculations, the merging process is simplified (See Figure 4.2). The redundant calculations are performed on the overlap regions. Use of the overlap regions means that the frontier information does not have to be examined when the spacing between the two strips is calculated. Care must be taken to ensure that the overlap is sufficient to account for all possible design rules. To determine the

spacing between two adjacent strips, only the calculated spacing of the overlapping grid lines must be compared. This is much simpler than comparing the pairs of grid points and frontiers of the edges of the strips.

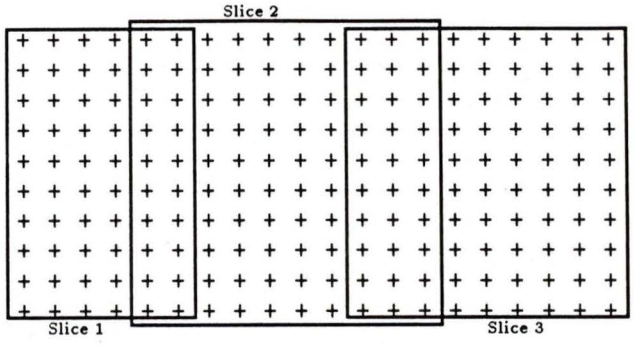


Figure 4.2: Overlapping Strips

To clarify the merging process, an examination is made of the sections of the spacing arrays of the two strips which overlap. The spacing array for the merged strips is generated by copying the non-overlapped entries and using the maximum spacing value of the entries which overlap. Figure 4.3 depicts this process.

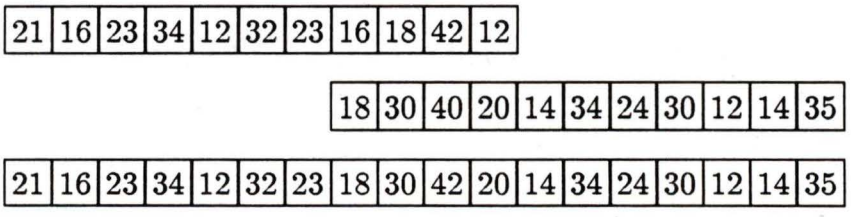


Figure 4.3: Merging Process

Of the two remaining steps to be selected, the vertical spacing step can now be performed since all of its input has been generated. The vertical spacing calculations can also be executed in parallel for separate regions of the layout. A partitioning

of the layout into horizontal strips gives the most efficient distributed calculation. These vertical strips will include all the grid points in the same horizontal grid line. As in the horizontal spacing case, if the strips slightly overlap then the merging process is simplified. The regions plotted for the horizontal spacing step are different from the ones required for the vertical spacing step. This means that if the vertical spacing calculations are to be performed in parallel then the regions must be replotted for vertical spacing. This replotting would not be necessary if all the compactor's steps were assigned to a single process. A separate process could be created containing the plotting and the vertical spacing steps but since vertical spacing depends on the results of horizontal spacing the vertical spacing step can be assigned to the same process as the horizontal step. This assignment means that the symbolic description has to be sent to the spacing process only once.

The remaining unselected step is the fleshing step. It requires the output of the parser, horizontal spacing and vertical spacing steps. The fleshing step produces a description of the mask geometry which must be saved on disk. Since the fleshing step can be performed in parallel, a decision on distributing the step depends on the cost of merging the output from each fleshing process to the processor containing the disk.

In summary, the distributed compactor is a system composed of two types of processes. Figure 4.4 depicts this system. One type of process includes the parsing and fleshing steps and additional steps for collecting and distributing information to the other processes in the system. The second type of process contains the plotting, net generation, horizontal spacing and vertical spacing steps. A process of the second type is created for every processor in the hardware configuration of the local area network. The degree of parallelism is directly proportional to the

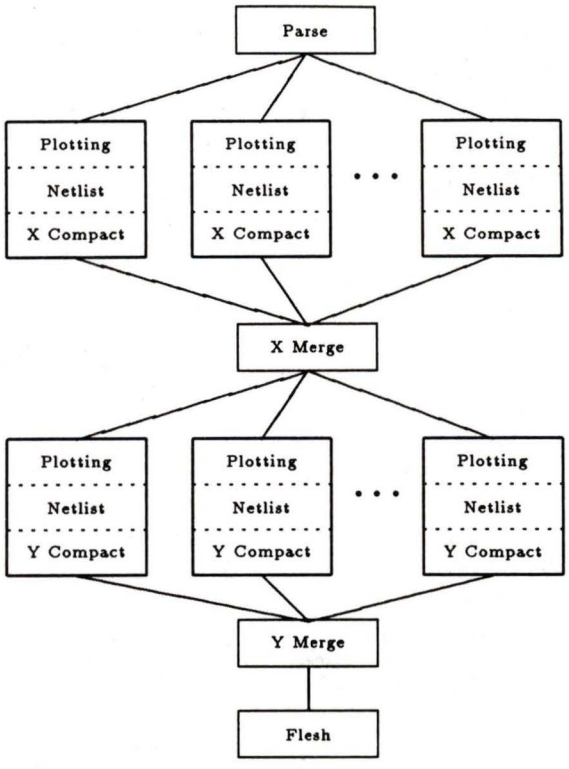


Figure 4.4: Compactor Block System

number of processes of the second type. The structure of the system resembles the client/server model, where the first type of processor is the client and the second is the server.

When a client process is invoked to compact a layout, the client:

1. parses the symbolic description,
2. assigns a vertical strip of the layout to each of the available servers,
3. waits for each of the server processes to compute the horizontal spacing for its assigned strip,
4. receives the spacing calculation for each vertical strip and merges the horizontal spacing to produce the x grid,

5. sends the x grid and the horizontal strip description to each of the servers,
6. waits for each server to calculate the spacing of its assigned horizontal strip,
7. receives the spacing calculation for each horizontal strip and merges the vertical spacing to produce the y grid,
8. and finally, the x grid, y grid, and layout description are used to produce the mask information.

Steps 1, 2, 5, and 8 are performed by the client and steps 4, 5 and 6, 7 are executed by the servers.

4.5 Implementation of the Compactor with SUN's RPC

The implementation of the distributed compactor in a local area network requires a run-time system which provides inter-process communication and multi-tasking facilities. The compactor was developed using the SUN 3.2 operating system, a variation of 4.2/3BSD UNIX and SUN's RPC (Remote Procedure Calls). The compactor runs on any machine which supports 4.2/3BSD IPC primitives and SUN's RPC system.

Remote procedure calls provide the means for one process to communicate with other processes on the same machine and other machines connected to the network. An RPC has the same semantics as a regular procedure call with the exception that the procedure is executed in a separate process. This distinction means that the arguments and returned results of the remote procedure must be sent between the process which calls the remote procedure and the process which executes it. SUN's RPC implementations use 4.2/3BSD UNIX Inter-Process Communication

(IPC) facilities. Processes containing remote procedures are usually referred to as server processes and processes invoking these remote procedures are called client processes. The exact semantics for an RPC are found in section 2.7.2.

Unfortunately the semantics of an RPC do not allow for parallel execution of remote procedures on different machines. In SUN's implementation the application programmer has control of the communication mechanism and thus can change the semantics of remote procedures. For the parts of the compactor that are to be executed in parallel the new semantics for an RPC are:

1. The client sends a message containing information as to which remote procedure is to be executed and the remote procedure arguments.
2. The server processes send a message back acknowledging the call.
3. The remote procedure is then invoked.

After step 2 the client process's remote procedure call returns, allowing the client and server to execute in parallel. Obviously no results are returned by this remote procedure call. There are two techniques for returning the result of the procedure to the calling process: 1) the client can poll the server or 2) the server process can send a message to the client when it is done.

The modified RPC can be used to start the parallel execution of parts of the compaction algorithm at different servers. The client does this by calling the remote procedure located at the different servers in turn one after another. For this technique to make sense, the time spent executing the remote procedure must be greater than the time required to send the calling message. This calling technique also means that the execution of the remote procedure at each server will be staggered in time.

Procedure	Type	Step Number
construct_table	regular	1
assign_work_orders	remote	2
init_compact_server	remote	2
assign_steps	regular	2
start_x_compact	remote	2,3,4,5
merge_x	regular	5
start_y_compact	remote	5,6,7
merge_y	regular	7
flesh	regular	8

Table 4.4: Procedure List

The implementation of the distributed compactor uses the regular and modified RPC. The modified RPC provides the only means of parallel execution for parts of the distributed compactor. Table 4.4 outlines the main procedure used in this implementation of the distributed compactor, how these procedures are related to the compactor steps as outlined in the last part of section 4.4, and if they are remote procedure calls. The parts of the compactor implemented by these procedures are also given with their step numbers. These step numbers refer to steps give in section 4.2. The remaining sections in this chapter will discuss this implementation.

4.5.1 construct_table

The construct_table routine parses the layout description and produces the internal representation for the layout. This procedure implements the parsing step and calculates the bounding box of the given layout. Since parsing is serial in nature this procedure is executed only by the client process.

4.5.2 assign_work_orders

The number of server processes available is determined by this step. This is accomplished by performing a broadcast remote procedure call. Normally when an RPC is invoked the remote procedure number and processor are specified. In a broadcast RPC only the remote procedure is specified and all processors which contain that particular server will respond. The servers which respond to the broadcast RPC are marked as available.

Broadcast RPCs can be implemented with 4.2/3BSD UDP (User Datagram Protocol). Unfortunately this protocol was found not to be reliable enough for the communication required by the distributed compactor. The unreliablity of a UDP socket can be attributed to the lack of system buffers to accept incoming packets. Connections were then established using SOCKET STREAM protocol between the client and all the available servers. A major disadvantage of the SOCKET STREAM protocol is that it does not implement any broadcast facility, so any information which could be broadcast from the client to the server has to be sent to each server separately.

The final task of the assign_work_order procedure is to use the number of available servers to determine how the layout is partitioned among the set of servers. The partitioned regions produced are overlapped to simplify the merging process.

4.5.3 init_compact_server

Init_compact_server is a remote procedure which transmits the layout description to a server process. This procedure uses the connection established by assign_work_order and thus has to be invoked for each server. Ideally, transmitting the layout description to each server could be done using a broadcast protocol but the broadcast

service available proved to be unreliable and thus unusable as stated above. The typical amount of information transmitted by this step is given in Table 4.1.

4.5.4 assign_strips

The assign_strips used the regular RPC call to assign the layout partitions to each of the servers. The procedure is called for each of the available servers in turn. Since the information sent to each server is small the amount of parallelism lost due to the sequential assignment is negligible.

4.5.5 start_x_compact

The start_x_compact procedure is responsible for starting the parallel execution of the horizontal compactions. It then waits for each server to finish and collects the spacing results. Before the start_x_compact routine is called each server has all the data it needs to begin the horizontal compaction. This compaction is started by using the parallel RPC.

To determine when the horizontal compaction has been completed by all servers, the client process sets up a regular RPC that is invoked by each of the servers. The client waits until it has been called by all of the servers. Start_x_compact executes the following sequence.

1. Setup a remote procedure (finish_compact) that will be executed by the client process.
2. For all of the available servers, invoke the parallel RPC to start the horizontal compaction.
3. Wait until all of the servers have called finish_compact.

4. For each server invoke the regular RPC routine which returns the spacing calculations for each server.

In this procedure the roles of the client and set of servers are reversed so that the client can determine when all the server processes have completed the horizontal compaction.

To perform the horizontal compaction each server process has to perform the plotting, net generation, and horizontal spacing calculation for its assigned region.

4.5.6 merge_x

This routine combines the results of each of the servers to give the complete horizontal spacing.

4.5.7 start_y_compact

The start_y_compact procedure behaves exactly the same as the start_x_compact procedure except vertical compaction instead of horizontal compaction is performed and the complete x spacing is transmitted to each server. The compact_finish routine is again used to indicate when all the servers have finished the vertical compaction.

4.5.8 merge_y

This routine is executed by the client process to combine the vertical spacing results. It also uses the merging process outlined in section 4.4.

4.5.9 flesh

Flesh is the final procedure executed by the client to produce the mask data files from the horizontal and vertical spacings and the layout description.

Chapter 5

Tests and Measurements

5.1 Introduction

The purpose of the chapter is to present the measurements taken of the runtime behaviour for the distributed compactor and an evaluation of those measurements. These measurements were performed to determine the runtime characteristics of the distributed compactor when the input problem size, the makeup and the hardware configuration on which the system was executing were varied. Another reason for performing these measurements was to determine the effectiveness of the partitioning of the compaction algorithm.

The symbolic layout input to the compaction system and the run-time configurations were the two independent parameters in the measurement of the system. The symbolic layouts used consisted of layouts ranging from small cells to the major modules of VLSI chips. The compaction system executed under two different hardware configurations, a network of SUN 3 workstations and a network composed of three microVAXIIs, a VAX750, a VAX780, and a SUN3. In each of the two

configurations, subsets of the network were also used.

The timings for the distributed compactor were compared with those produced by VIVID's 1.3 HCOMPACT for the same symbolic input.

The first part of this chapter deals with what was measured and how these measurements were taken. The second part of the chapter discusses conclusions drawn from the measurements collected.

5.2 Layout Cell Test Cases

The layout cells listed in Table 5.1 were used to test and evaluate the distributed compactor system. The layouts chosen provide a reasonable mix of layout sizes and complexities. A brief description of each of the cells follows.

nand_4 a 4 input fully complementary CMOS nand gate.

nand_8 a 8 input fully complementary CMOS nand gate.

ff a multivalued logic T-gate cell [46].

tt(2-20) variable sized PLA's (2 to 20 input and output lines).

shift_30 a 30-bit barrel shifter.

bit a bigger barrel shifter.

mem a 64-by-8 bit static ram block.

5.3 Measurements

Runtime measurements consisted of the amount of real, user and system time consumed by a specific activity. The UNIX process accounting system calls were used

cell name	number of primitives	matrix height	matrix width	plot number
nand_4	12	7	12	1
nand_8	20	9	20	2
ff	355	23	37	3
tt2	396	31	27	4
tt3	645	55	35	5
tt4	919	61	44	6
tt5	1233	67	53	7
tt6	1603	73	62	8
tt10	3428	79	99	9
tt20	11158	139	189	10
shift_30	8550	106	181	no
bit	23275	176	295	no
mem	31768	282	237	no

Table 5.1: Symbolic Layout Test Cases

to perform the timing measurements. Real time refers to the amount of wall clock time required by the activity. User time is the amount of CPU time that an activity used. Only the user time for equivalent processors can be directly compared. System time is the amount of CPU time that the operating system used servicing requests generated by the activity. If the activity being measured were the only one performed by the CPU, then real time should be the same as user time and system time added together. All times are given in seconds.

The raw data collected for one instance of the distributed compactor is given in Figure 5.1. This instance consisted of four processing nodes; one node, arlene, contained the client process, while the other three nodes, jon, lyman, and pooky contained copies of the server process.

The information contained in this figure can be grouped into three sections. The first section, the top three lines, tells which symbolic layout was compacted,

```
Filename    = mem
x_len = 236  y_len = 281
num of primitives = 31768

            Times per Node
                   jon          lyman          pooky
            ---------------------------------------
totalclient =  2237.18
parser      =  928.57
pingcom              0.38           0.55           0.64
layoutcom           86.63          80.69          80.27
assigncom            0.03           0.03          19.92
xstartcom            0.02           0.02           0.03
xcompact           128.64         294.84         258.55
xcompactwait = 295.25
xresultcom           0.06           0.07           0.07
ystartcom            0.08           0.10           0.15
ycompact           233.04         572.15         514.06
ycompactwait = 572.28
yresultcom           0.06           0.07           0.08
fleshing     = 170.36

Serial = 1098.931422 [49.121369%]
Communication = 269.948376 [12.066480%]
Parallel = 867.529569 [38.777889%]
Parallel Speed Up = 2.306872
Total Speed Up = 1.506435
```

Figure 5.1: Raw Data

the number of primitives and the size of the symbolic layout. The next section gives the timing information gathered. The last section summarizes the raw data.

For the middle section all the lines with only one number and an equals sign contain the timing information for the client process. The lines with three numbers provide the timing information for each of the server processes. The heading preceding this section associates the columns in the server timing information with a particular processing node.

Each line in the timing information gives the timing data for relevant parts of the distributed compactor. The order of the lines is the same order as the sequence in which the measured events occur. The meaning of each line is given by the following list.

totalclient measures the real time taken to compact the input symbolic layout. In other words, this is the amount of time that the user has to wait for the compaction to finish. The total of all the timing data for the parts of the compaction should be equal to the total time.

parser is the time taken to convert the textual form of the symbolic layout into the internal form used by the compactor (i.e. it measures the time for the parsing step).

pingcom measures the time taken for the server processes to respond to the broadcast RPC used to determine the available servers. The timing value for each node is the time taken for that server to respond to the broadcast RPC including the time taken to call the RPC. The total time spent in the pingcom event is the time for the last node to respond.

layoutcom is the time taken to communicate the symbolic layout information to each server process. The individual timings add to give the time needed to send the symbolic layout list to all servers.

assigncom measures the time required to tell each server which part of the layout it will compact. The total time for this event is again the total of the individual times.

xstartcom measures the time for a parallel RPC to initiate the horizontal compaction at each server. The total time is also the sum of the individual times.

xcompact measures the real time required at each server process to finish the assigned horizontal compact.

xcompactwait is the amount of time that the client spends waiting for all of the horizontal compactions to finish.

xresultcom measures the time required by the client to collect the results of the horizontal compaction from each of the servers. The total time for this event is the sum of all the individual times.

ystartcom is the time required to execute the parallel RPC to start vertical compaction on all the servers. The total time is also the sum of the individual times.

ycompact measures the real time required at each server process to finish the assigned vertical compaction.

ycompactwait is the amount of time that the client spends waiting for all of the vertical compactions to finish.

yresultcom measures the time required by the client to collect the results of the horizontal compaction from each of the servers. The total time for this event is the sum of all the individual times.

fleshing measures the time for the client to perform the fleshing step.

These timing results can be divided into three categories: serial, parallel and communication. The serial category contains the events which have to be executed sequentially by the client, excluding any communication events. The parsing and fleshing events compose the serial category. The communication category contains all the events which contain communication (i.e. RPCs) between the client and server. All the timing events containing the suffix *com* are in the communication category; xcompact and ycompact make up the parallel category. The total of

the events in the serial category places a bound on the amount of speedup improvement which can be achieved by distributing the compaction algorithm since distributing the computation will not affect the computation in the serial category. The communication category gives the cost involved in distributing the algorithm. The computation performed in the parallel category provide the speedup in the execution time of the compactor.

The first three lines of the last section of Figure 5.1 give the time and percentage that the three categories contribute to the execution time of the compactor. The next line gives the parallel speed up, which is defined as the ratio of sum of the x and y compact times to the sum of time waiting for the x and y compacts to finish. The last line gives the total speed up which is defined as the ratio of the sum of the time taken by the servers and client to the total client time.

5.4 Summarized Measurements

The preceding section detailed the measurements taken for a single execution of the distributed compactor. The behaviour of the compactor was studied in more detail as a function of the input problem and number of processors involved. The detailed information is summarized into serial, parallel, and communication categories and the results are tabulated for a series of tests as shown in Tables 5.2a and 5.2b.
Table 5.2a contains an entry for each test run, giving the number of processors used, total real time, the amount of time for each category, cpu usage for the client and a note number which refers to the matching entry in the second table, Table 5.2b. Cpu usage is the percentage of the real time that client was being run by the processor. The usage parameter is important in interpreting the performance of the compactor,

Layout:tt4 Client: arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	125.94	25.39 [20%]	92.51 [73%]	7.36 [6%]	79.8	1
1	136.10	26.38 [19%]	105.15 [77%]	3.96 [3%]	85.9	2
1	123.05	28.09 [23%]	90.87 [74%]	3.53 [3%]	83.8	3
2	123.01	42.14 [34%]	66.34 [54%]	12.99 [11%]	54.2	4
2	152.49	30.18 [20%]	111.51 [73%]	9.79 [6%]	73.6	5
2	99.92	29.55 [30%]	62.75 [63%]	6.81 [7%]	81.0	6
3	99.80	24.84 [25%]	61.70 [62%]	12.11 [12%]	82.2	7
3	108.36	26.93 [25%]	67.14 [62%]	13.30 [12%]	75.3	8
3	222.45	23.62 [11%]	95.74 [43%]	102.12 [46%]	24.1	9

(a)

Note Number	Processors [cpu usage]		
1	irma [97.5%]		
2	lyman [87.9%]		
3	irma [99.0%]		
4	lyman [89.0%]	irma [94.2%]	
5	irma [87.3%]	pooky [86.0%]	
6	lyman [93.0%]	irma [88.7%]	
7	irma [99.3%]	lyman [95.4%]	pooky [85.4%]
8	lyman [91.8%]	irma [92.5%]	pooky [79.9%]
9	irma [99.6%]	lyman [91.3%]	pooky [55.3%]

(b)

Table 5.2: Serial, Parallel, and Communication Summary

since it relates the system activity to the time taken for a compact to finish. The smaller the percentage the longer the user had to wait because the system was busy and not performing the compaction. For larger problem sizes the compaction process is delayed due to the paging caused by the large memory requirement. The distributed compaction is, of course, executing on all the processors and therefore the cpu usage for each of the server processes is also important. The note number of the first table gives the entry in the second table which lists which processors the server processes were executing on and the cpu usage for each of the servers. Cpu usage is calculated by dividing the cpu time spent in executing the process by the real time.

5.5 Testing and Results

To perform the runtime test on the distributed compactor the symbolic layouts described in Table 5.1 were used. The distributed compactor was primarily tested using three different network configurations. These configurations were:

1. A VAX 11/750 (arlene) as client and three MicroVAX II's (irma, lyman, and pooky) as servers.
2. A SUN 3/160 (jon) as client and three MicroVAX II's (irma, lyman, and pooky) as servers.
3. A set of seven SUN 3/50 and SUN 3/110 where one of the seven acted as the client and the remaining 6 acted as servers.

The server processors for the three configurations are all of the same processor type. The distributed compactor currently assigns equal work to each server process. Therefore, if the server processes were run on processors with differing processing power, then the more powerful processors would not be utilized efficiently since they

cell name	num. of primitives	Arlene (u+s) [r]	Jon (u+s) [r]	Suna (u+s) [r]	Lyman (u+s) [r]
nand_4	33	6.89 [10.44]	1.52 [2.06]	2.18 [5.00]	4.18 [6.60]
nand_8	62	12.33 [15.32]	2.74 [3.32]	3.26 [4.80]	7.54 [8.53]
ff	355	62.42 [67.27]	13.92 [16.16]	14.42 [18.18]	38.17 [49.16]
tt2	396	63.7 [69.28]	14.28 [15.48]	14.32 [18.04]	38.08 [41.46]
tt3	645	131.57 [150.24]	28.86 [30.06]	29.68 [33.27]	78.17 [82.62]
tt4	919	186.44 [230.39]	40.92 [41.96]	42.08 [46.28]	109.13 [116.28]
tt5	1233	248.54 [259.97]	55.1 [56.16]	56.70 [58.90]	148.8 [157.97]
tt6	1603	318.5 [325.50]	71.12 [72.48]	74.06 [77.48]	192.29 [211.74]
tt10	3428	632.66 [664.18]	141.1 [143.22]	146.26 [151.03]	383.18 [410.36]
tt20	11158	2194.10 [2714.15]	482.12 [490.00]	496.52 [511.67]	1307.96 [1375.96]
shift_30	8550	1608.46 [1800.25]	358.32 [365.72]	367.10 [374.34]	979.95 [1065.15]
bit	23275	4459.66 [4951.54]	1219.34 [3335.56]	1329.5 [4008.39]	2727.05 [3364.64]
mem	31768	5331.4 [5657.18]	1244.74 [1405.88]	1294.62 [1491.60]	3250.35 [3693.77]

Table 5.3: Serial Times

would remain idle while the less powerful processors finished their assigned work. Test cases performed with server processes running on different processors showed this to be true.

Table 5.3 gives the sum of the user and systems times in seconds, as well as the real time for a serial version of the compactor with the sample layouts. The real time is enclosed with "[]". This table also provides benchmark times with which to compare the distributed compactor. The relative processing power of each of the processors can also be derived from Table 5.3. Relative to Arlene (the VAX 11/750), Jon is a factor of 4.4 faster, Suna by 4.2 and Lyman by 1.6. With the exception of the last four rows in Table 5.3 and Jon's timings, the data in the table is plotted in Figure 5.2. This plot depicts the fact that the run time is almost linear with respect to the number of primitives. Jon's timings were not plotted since Suna'a timings were essentially the same.

Compaction times for the sample layouts were collected for each configuration. The number of servers used by the configurations was varied from one server to

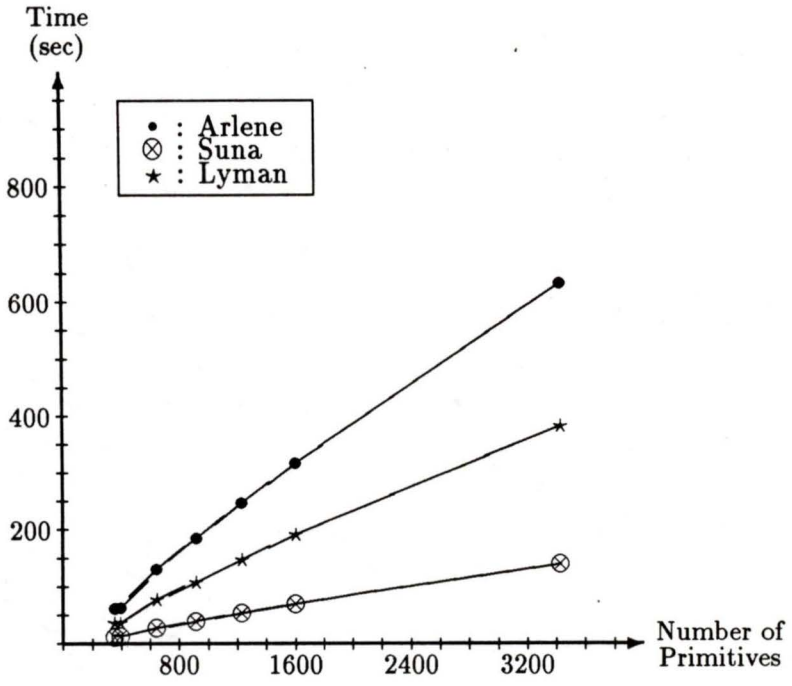


Figure 5.2: Serial Compaction Times

all available servers. For all tests the compaction was performed when the systems were lightly loaded. Appendices B, C, and D contain the timing results, presented in the same format as Table 5.2, for the three hardware configurations.

The results for the configurations of arlene as the client and irma, pooky and lyman as servers are summarized in Table 5.4. The rows in the table correspond to the time, in real seconds, for each input problem. Times are given for the serial compactor, Vivid's compactor and the distributed compactor. The serial compactor and the vivid compactor were executed on arlene. The distributed compactor column is the best time from among all the distributed times. The number in "()" is the number of processors used for that time. The ratio of the serial time to the distributed time is contained in the last column of the table. The timing results show a speed improvement of 1.1 to 3.0. Although this seems like a nice result, to get more meaningful speed ratios the ratios have to be scaled by the processing

cell name	num. of primitives	Serial Compactor	Vivid Compactor	Best Distributed	Ratio S/D
nand_4	33	10.44[66%]	82.3[83%]	7.38(1)	1.4
nand_8	62	15.31[81%]	169.1[85%]	12.68(2)	1.2
ff	355	67.27[93%]	981.9[81%]	38.15(2)	1.7
tt2	396	69.29[92%]	923.9[82%]	38.92(2)	1.6
tt3	645	150.24[88%]	1823.0[78%]	65.13(2)	2.3
tt4	919	230.38[80%]	3959.1[58%]	99.8(2)	2.3
tt5	1233	259.96[96%]	5004.8[67%]	127.63(2)	2.0
tt6	1603	325.49[98%]	7539.0[60%]	169.44(3)	1.9
tt10	3428	664.18[95%]	12902.4[89%]	280.82(2)	2.3
tt20	11158	2714.14[81%]	83620.0[70%]	899.54(2)	3.0
shift_30	8550	1800.25[89%]	42693.1[88%]	1153.77(3)	1.6
bit	23275	4951.56[90%]	—	4169.94(1)	1.2
mem	31768	5657.17[94%]	—	5361.69(2)	1.1

Table 5.4: Arlene as Client

power ratio of VAX 11/750's and microVAXII's (i.e 1.6X). The reason for applying the scaling is that if microVAXII's were available to act as servers they could also just execute the serial version of the compactor. The speed ratios are then reduced to 0.7 to 1.9.

In the next configuration a fast client (jon) is matched with three slower server processors (lyman, pooky, and irma). Table 5.5 lists the results. This configuration is not a very good assignment of the processors to the problem. The configuration was tried only because it was available. It is immediately obvious that the speed up ratios with one exception are all less than 1.0. This test configuration does show how sensitive distributed algorithms are to hardware configuration with non-uniform processor power.

The one exception is interesting in that it points out the advantage of this distributed system with respect to memory usage per processor. The long time taken for **bit** to compact is due to the high number of page faults that occur

cell name	num. of primitives	Serial Compactor	Vivid Compactor	Best Distributed	Ratio S/D
nand_4	33	2.06[73%]	18[85%]	4.0(1)	.51
nand_8	62	3.32[83%]	38[91%]	6.74(3)	.49
ff	355	16.16[86%]	200[98%]	21.76(3)	.74
tt2	396	15.48[92%]	193[97%]	22.84(3)	.68
tt3	645	30.06[96%]	355[98%]	42.66(3)	.70
tt4	919	41.96[98%]	555[99%]	60.58(3)	.69
tt5	1233	56.16[98%]	811[98%]	80.64(3)	.69
tt6	1603	72.48[98%]	1159[98%]	101.60(3)	.71
tt10	3428	143.22[98%]	4078[76%]	191.80(3)	.75
tt20	11158	490.00[94%]	102390[24%]	615.38(3)	.80
shift_30	8550	365.72[98%]	48936[30%]		
bit	23275	3333.56[36%]	—	1304.22(3)	2.56
mem	31768	1405.88[89%]	—		

Table 5.5: Jon as Client

with this design. When the design is partitioned the same paging behaviour is not observed.

The last configuration studied consists of a uniform network of SUN3 workstations. The SUNs configuration is a more typical network configuration. In the test performed, the network consisted of seven Suns with one acting as client and the remaining six acting as servers. The results are shown in Table 5.6. With the exception of the **bit** layout the speedup ratio varies from 1.0 to 2.3 with an average speedup of 1.7. Table 5.7 gives the average serial time as a percentage, the maximum speed up, and the realized speed up for each layout, all of which are derived from the data in Table 5.6.

Figures 5.3 and 5.4 condense the tabulated data in appendix C to 13 graphs. The times measured in Figures 5.3 and 5.4 give the real time (clock time) for test layouts. The real time is affected by system activity. Figures 5.5 and 5.6 give the cpu time required (both system and user) for these layouts. The timing information for the

cell name	num. of primitives	Serial Compactor	Best Distributed	Ratio S/D
nand_4	33	5.00 [44%]	3.62(1)	1.4
nand_8	62	4.80 [68%]	4.78(1)	1.0
ff	355	18.18 [80%]	12.2(3)	1.5
tt2	396	18.04 [80%]	12.0(4)	1.5
tt3	645	33.27 [89%]	21.2(4)	1.6
tt4	919	46.28 [90%]	28.6(5)	1.6
tt5	1233	58.90 [96%]	39.0(4)	1.5
tt6	1603	77.48 [96%]	43.2(6)	1.8
tt10	3428	151.03 [97%]	74.0(6)	2.0
tt20	11158	511.67 [97%]	224.3(5)	2.3
shift_30	8550	374.34 [98%]	182.6(6)	2.1
bit	23275	4000.39 [33%]	506.9(6)	7.9
mem	31768	1491.60 [87%]	747.5(5)	2.0

Table 5.6: Suna as Client

Cell Name	Percentage Serial Time	Maximum Speed Up	Actual Speed Up
nand_4	0.57	1.8	1.4
nand_8	0.47	2.1	1.0
ff	0.40	2.5	1.5
tt2	0.40	2.5	1.5
tt3	0.37	2.7	1.6
tt4	0.41	2.5	1.6
tt5	0.41	2.5	1.5
tt6	0.41	2.5	1.8
tt10	0.31	3.2	2.0
tt20	0.33	3.1	2.3
shift_30	0.39	2.6	2.1
bit	0.39	2.6	7.9
mem	0.36	2.8	2.0

Table 5.7: Result Summary

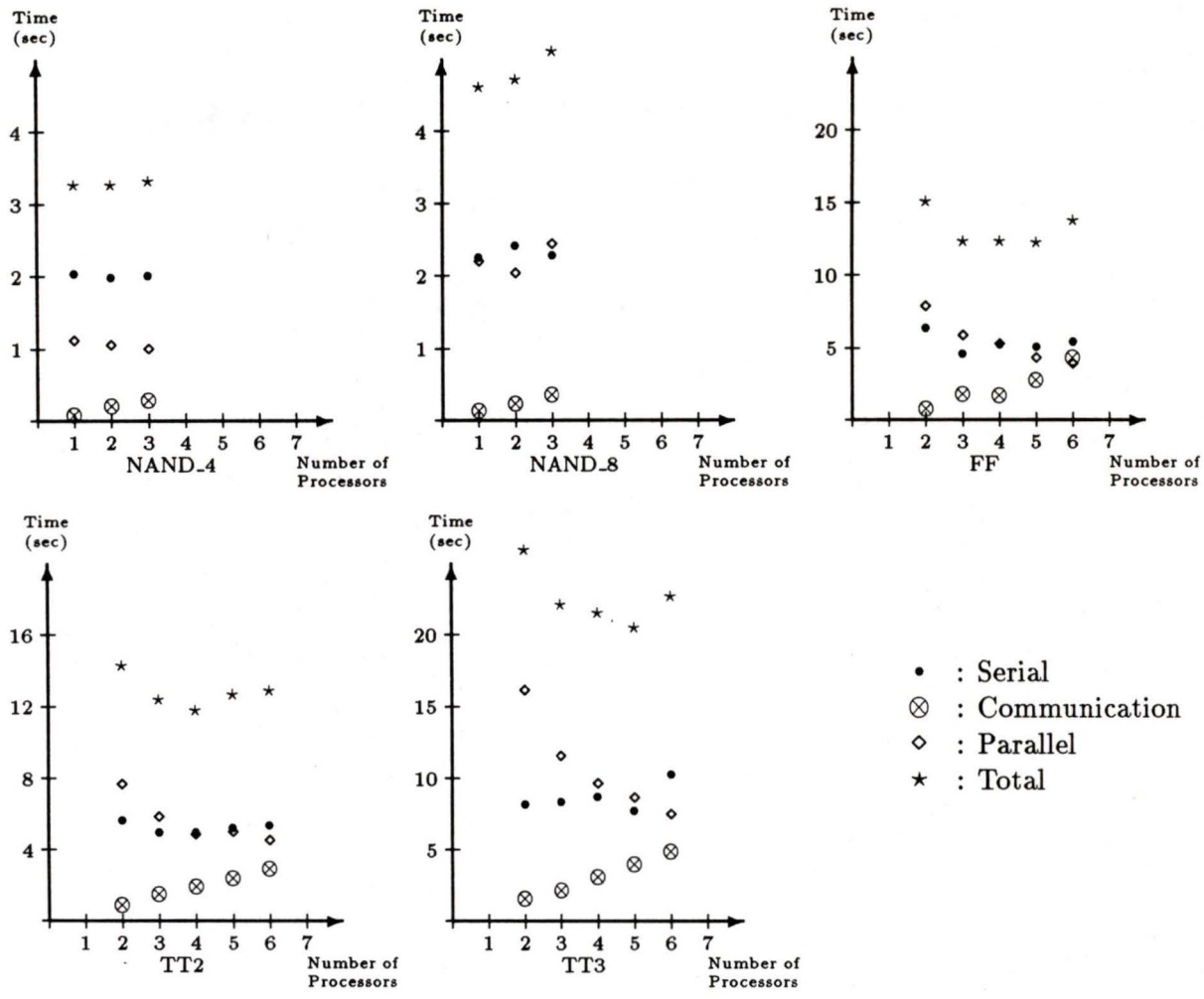


Figure 5.3: Real times for nand_4, nand_8, ff, tt2, and tt3

latter two figures give the time required for compacting the input layouts ignoring system activity and assuming that only the DLCS is executing. The graphs, one per input layout, show the average serial time, communication time, parallel time and total time for the configuration of seven Suns. The same trends are observed in the other two configurations.

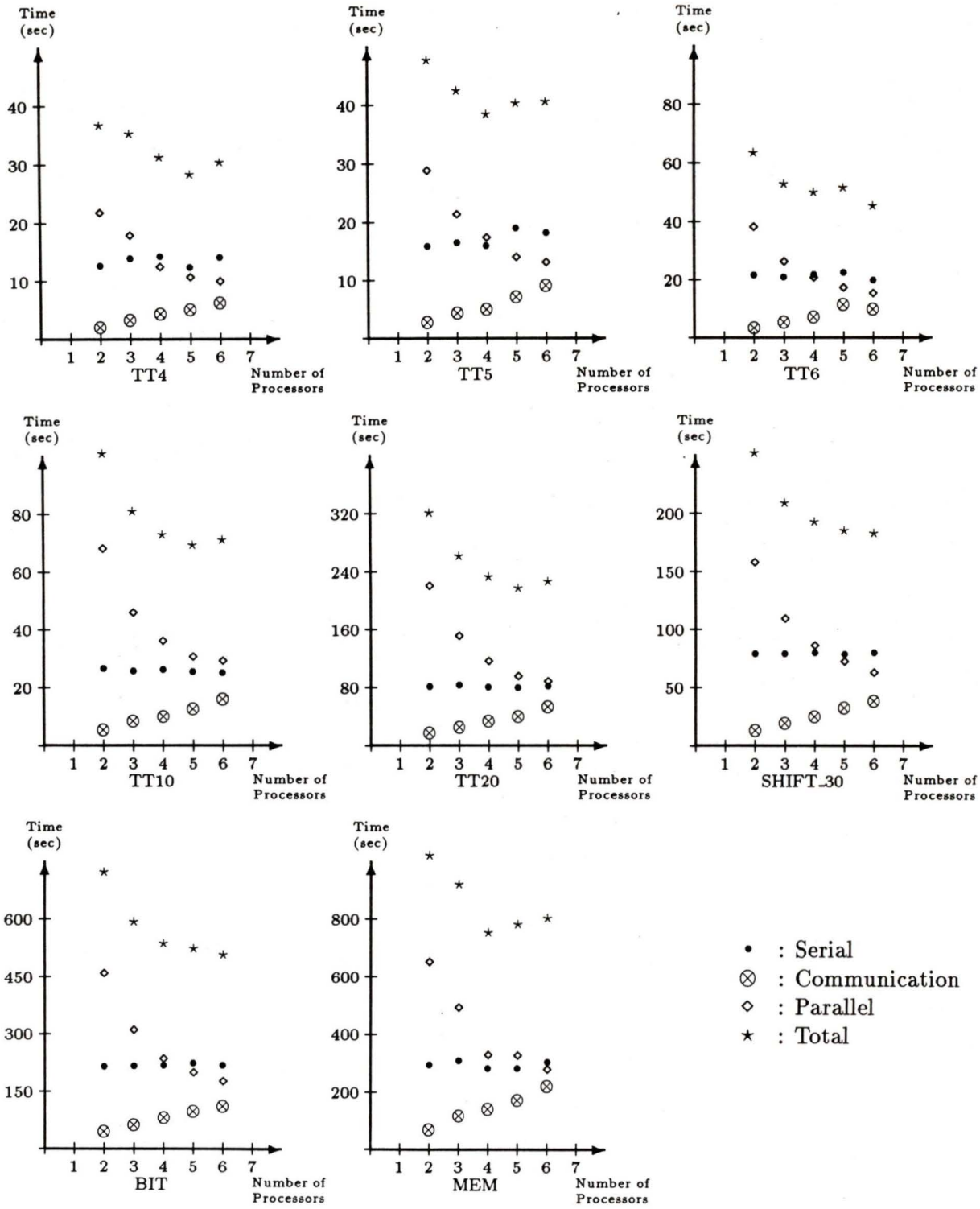


Figure 5.4: Real times for tt5, tt6, tt10, tt20, shift_30, bit, mem

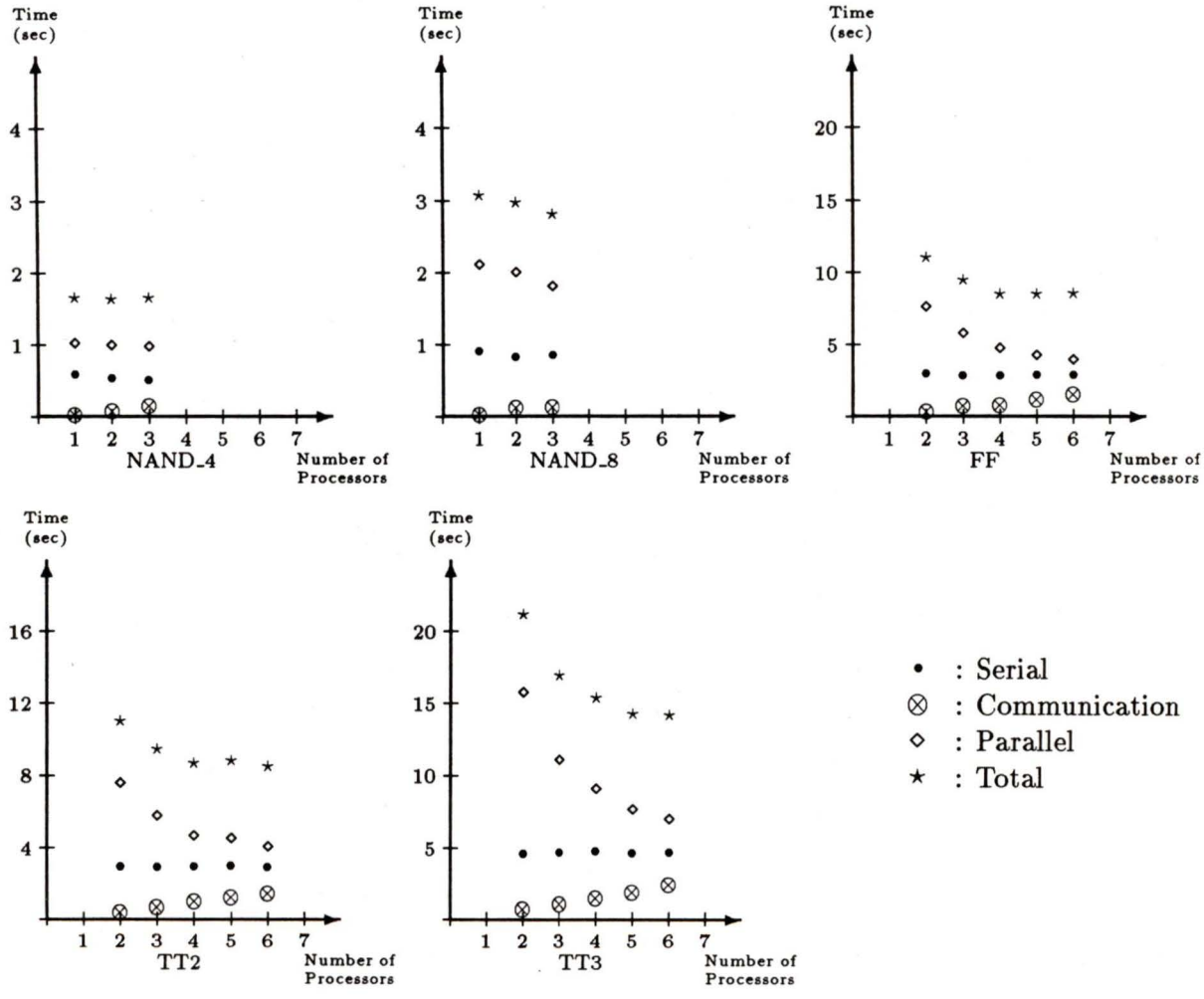


Figure 5.5: User+system times for nand_4, nand_8, ff, tt2, and tt3

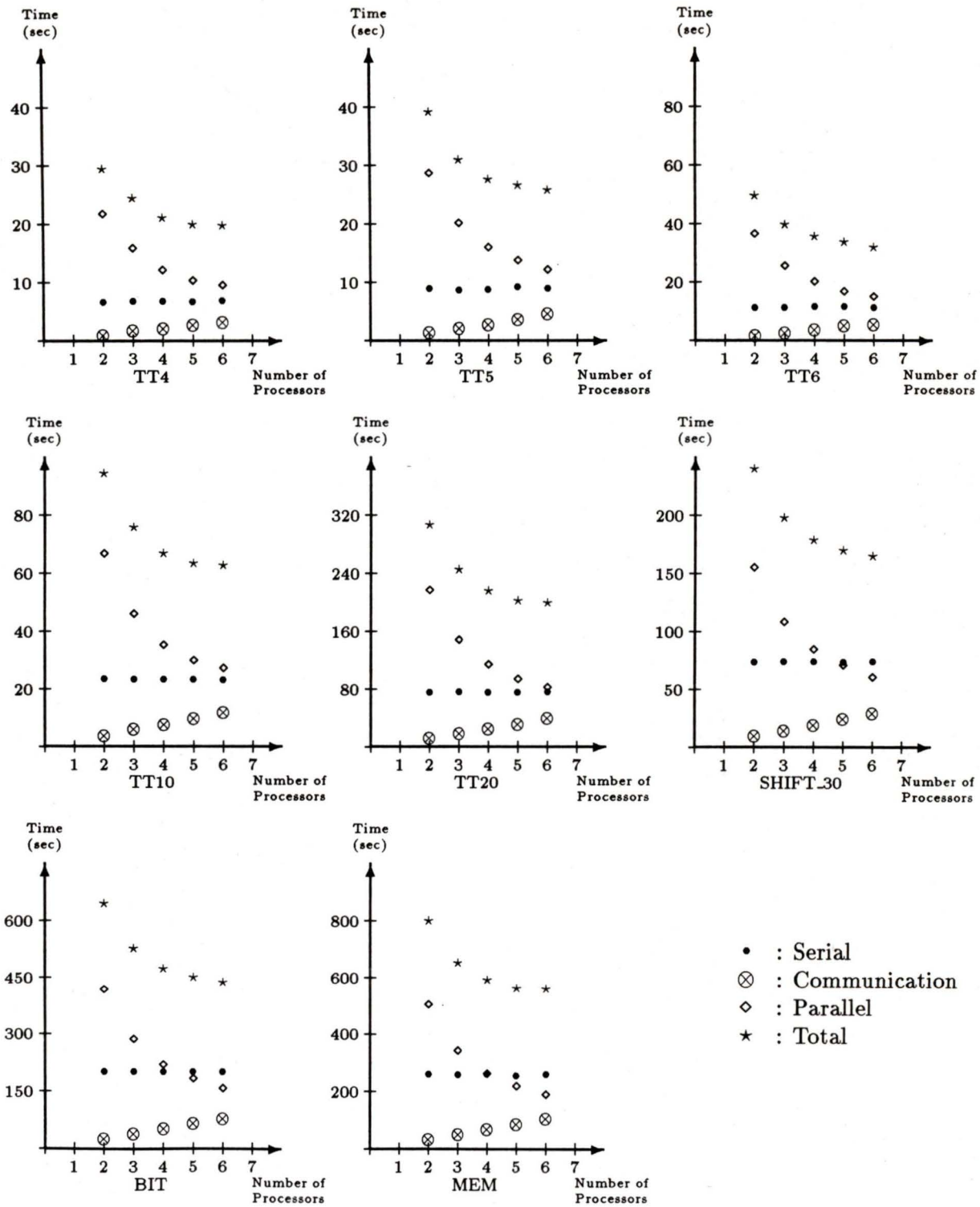


Figure 5.6: User+system times for tt5, tt6, tt10, tt20, shift_30, bit, mem

Chapter 6

Conclusions

The conclusions for this thesis can be divided into those dealing with the compaction algorithm and those dealing with the distribution of the algorithm. All conclusions for the distributed algorithm are based on the measurements presented in the previous chapter. The conclusions for the compaction algorithm are based on comparisons with VIVID's compactor and observations obtained during the design and implementation of the compaction algorithm. Finally, directions for further study are outlined for the design of the serial and distributed compaction algorithm.

6.1 Distributed Compactor

The performance of any parallel algorithm depends on the degree of parallelism inherent in the algorithm as well as on the processing system the algorithm is to use. Ideally, parallel processing with N processors will be N times faster than the equivalent serial system. For any algorithm which exhibits both serial and parallel behaviour the speed up factor can only be at best the reciprocal of the percentage of time taken for the serial calculations. This makes the assumption that the parallel

part of the algorithm can be performed by an infinite number of processors and thus require almost no real time. The speed up factor can be expressed as:

$$\text{speed up factor} = \frac{1}{\text{serial}}$$

where serial is the percentage of time spent in the serial part of the compactor. For example if 100 seconds of cpu time are required to compact a layout and the serial percentage is 40% then the algorithm will required at least 40 seconds of real time to finish. This assumes that the remaining 60 seconds of cpu time, which can be executed in parallel, is allocated to a large number of processors and the real time required is 60/N, where N is the number of processors. As N increases the amount of real time decreases. The compaction algorithm developed in this thesis does possess a serial nature so that the overall speed up is bounded.

Since the actual speed up is calculated by dividing the best distributed time by the serial time and the maximum speed up is derived from the serial time of the distributed compactor, the exact meaning of the two speed up factors is different. However, the factors are roughly comparable since the algorithm for the serial and distributed compactor is exactly the same. The actual speed up is a comparison between the actual distributed and serial compactions. The maximum speed up is an indication of the best possible result assuming the time taken for the parallel and communications parts is reduced to zero.

Table 5.7 does show that the serial time plays an important part in the overall performance of the distributed compactor. An obvious step to improve performance is to see if the serial parts of the compactor can be rewritten to improve their efficiency.

When Figures 5.3 and 5.4, which depict the run time behaviour of the compactor for individual layouts, are examined, the following trends are apparent:

1. As the number of processors increase, the amount of time spent in the parallel computation decreases almost linearly, up to a certain point, and then levels off to a constant.
2. Communication time increases linearly with the number of processors.
3. The serial time is, of course, a constant.
4. The total time taken for a compaction decreases until the effects of the decrease in parallel computation can no longer compensate for the communication time increase.

To increase performance with the addition of more processors the behaviour of the communication part of the compactor has to be modified. This is an area which requires further study; several directions are suggested later in this section. The collection of graphs also suggests that there is a minimum amount of work to assign (i.e. size of layout) to each of the servers. If too little work is assigned the communication cost is not offset by the time saved in parallel computation. The graphs for **nand_4**, **nand_8**, **ff**, and **tt2** illustrate that, if the problem is too small, the amount of time saved in the parallel calculation is minimal, and there is very little decrease in parallel execution as the number of servers (i.e. processors) is increased. As the problem size increases, the amount of time spent in parallel calculation decreases almost linearly as the number of processors increases. From the graphs it can be seen that this linear decrease levels off after four or five processors. This leveling off can be attributed to the constant overhead involved with the compaction algorithm.

The timing results for compaction of the layout **bit** shows one of the advantages of a distributed compactor over a serial one. The serial compaction for the **bit** layout required 2.5 times more time to compact than the larger layout **mem**, but for the distributed compaction **mem** took longer than **bit**. Since **mem** is a bigger

layout it should take more time to compact. **Bit** took longer in the serial case because of the large amount of paging that was required to compact this layout. In the distributed case, the memory requirements were distributed over the servers and thus the same paging behaviour did not occur. In general, the distributed compactor requires less memory per processor since only strips of the entire layout are plotted. This reduced memory requirement means that the server processor will in general require less paging and thus execute faster. This may be a very important consideration for larger designs.

Of the three hardware configurations studied, the network of SUN 3's gives the best distributed performance. The network where jon (SUN 3/160) acts as the client gives the worst performance. In designing a configuration, the faster processors should be used as the servers. The current assignment procedure assumes that all the server processors are of equal power; this procedure should be changed in a configuration where the processors have differing powers. Greater system throughput can also be realized if the assignment is also based on the current work load of the server processors. If a particular processor is heavily loaded, an assignment to that processor would delay the completion time.

To summarize, although the measurements taken for the distributed compactor show adequate performance increases with an increase in the number of server processes, the following areas have been identified as areas where performance gains can be obtained.

1. The algorithms and data structures used to parse the symbolic layout can be improved. Since most of the serial time is attributed to the parsing step, this should result in an increase in the maximum speed up factor. In retrospect the choice of the dynamics list in the parsing step was a poor one.

2. Communication time is a linear function of the number of servers since the

entire layout is transmitted to each of the servers. This communication strategy was necessary in order to guarantee data delivery. A reliable broadcast/multicast protocol or a partition of the transmitted layout so that each server only receives the relevant layouts should result in a decrease in communication time.

6.2 Serial Compactor

The benchmarks performed on the serial version of the distributed compactor show that the serial compactor is from 9 to 14 times faster than VIVID's HCOMPACT 1.2. The speed difference can be attributed to VIVID's use of VMR rectangles, which results in HCOMPACT requiring more memory to compact the same design. The extra memory requirement results in greater system overhead and more complexity in the spacing calculations. However when HCOMPACT is given a non-flattened design, HCOMPACT performs much better. HCOMPACT begins to be faster for layouts around the **tt10** size. This advantage would disappear if the DLCS compactor was converted into a hierarchical compactor.

Although HCOMPACT does produce slightly more compact designs its approach is also more complicated. The most important difference is the "grid breaking" of HCOMPACT. This allow some mask features to be placed closer together than otherwise possible. In the serial compactor, because there is only one virtual grid the spacing calculations are more constrained; this results in a larger layout.

During the design of the compaction algorithm the following observations were made about the virtual grid approach.

1. The design of the compactor is simplified if the layout editing system, which the designer uses to create the layout, produces the netlist and the well assignment for each layout primitive. Some of the current compaction systems do

the well assignment automatically, which sometimes produces well groupings which are not what the designer wants. A compaction system must allow the designer to specify a well assignment, but also to generate one if the designer does not.

2. Since the virtual grid approach associates a grid point with one electric net, device primitives are forced to occupy at least three grid points. This fragmentation of primitives complicates the spacing calculations.

3. An implicit assumption in the virtual grid approach is that the mask features generated around every grid point are roughly uniform in size. This assumption causes problems when dealing with big device sizes and causes the compactor to produce layouts which are bigger than necessary. One solution to this problem is to define more primitive layout symbols (i.e. pieces of a device).

4. The major asset of virtual grid compaction is that it eliminates the design rule analysis required by the other approaches. Corner rule testing is much simplified.

In summary, the serial compactor performs reasonably well, but some parts of the compactor implementation need to be reevaluated. The following list notes some possible changes to the implementation.

1. The current technique of specifying wells by tagging the primitives in a well is not sufficient. A new symbolic primitive for well specification should be defined, which would consist of a closed path around the well region.

2. The current method of using outlines for the spacing calculations should possibly be changed to a constraint method. This would allow use of the constraint graph technique, and hopefully produce smaller layouts. Constraint arcs could be generated instead of the generation of the outlines, as described in section 3.6.

3. To allow more flexibility in using different technologies, the system should have the capability to define new primitive symbols. For example, it should be possible to define a new symbol for a capacitor or a resistor.

4. The technology database is implemented as a set of routines and static declarations. A language should be defined for the design rule specification to allow easy specification of a new technology.

6.3 Future Directions

The following is a list of questions and topics which require more work.

1. An approach to parallelism which was not explored in this thesis is to partition the layout by the leaf cells. Each leaf cell would be assigned to a separate server processor. If there were more leaf cells than processors the compactor would then schedule cells to be compacted as the server processes finished.

2. One of the problems encountered with the runtime system was that reliable delivery of data from one process to a set of processes (i.e. a broadcast protocol) was not available. Primitives which do provide this service would have increased the performance of the DLCS system.

3. The DLCS system chose the virtual grid technique as the basis for the compactor; other approaches should be investigated. One approach which suggests itself is to combine a virtual grid layout with the constraint graph approach.

4. Since, at present the DLCS is is not very tolerant to failure of one of the the server processors, the area of fault tolerance should be studied. An example of one possible improvement is, if one of the server processors crashes, the client would pick another processor and restart the work that the crashed process was performing.

Bibliography

[1] J. W. Green, "Sticks compaction and assembly," *VLSI System Design*, June 1986, pp. 46–51.

[2] M. Schlag, Y. Z. Lioa, and C. Wong, "An algortihm for optimal two-dimensional compaction of VLSI layouts," RC 9739 (#42889), IBM, November 1982.

[3] T. Ohtsuki, ed., *Layout Design and Verification*, ch. Layout Compaction, pp. 119–235. *Advances in CAD for VLSI*, North-Holland, 1986.

[4] W. Wolf, "Sticks compaction and assembly," *IEEE Design and Test*, June 1986, pp. 56–63.

[5] W. Wolf, "An experimental comparison of 1-d compaction algorithms," in *Proceeding of the 1985 Chapel Hill Conference of VLSI*, (H. Fuchs, ed.), Computer Science Press, 1985, pp. 165–180.

[6] Y. E. Cho, "A subjective review of compaction," in *Proceedings of the 22nd Design Automation Conference*, June 1985, pp. 396–404.

[7] J. Rosenberg, "Chip assembly techniques for custom IC design in a symbolic virtual-grid environment," in *1984 Conference on Advanced Research in VLSI, M.I.T.*, January 1984, pp. 213–225.

[8] N. Weste, "Virtual grid symbolic layout," in *Proceedings of the 18th Design Automation Conference*, 1981, pp. 225–233.

[9] N. H. Weste, "Mulga: an interactive symbolic layout sytem for the design of integrate circuits," *Bell Systems Technical Journal*, vol. 60, no. 6, July-August 1981, pp. 823–857.

[10] N. Weste and B. Ackland, "A pragmatic approach to topological symbolic IC design," in *VLSI 81*, (J. P. Gray, ed.), Academic Press, 1981, pp. 117–129.

[11] D. G. Boyer and N. Weste, "Virtual grid compaction using the most recent layers alogrithm," in *Proceedings of the International Conference on Computers-Aided Design*, 1983, pp. 92–93.

[12] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *CACM*, vol. 25, no. 10, October 1982, pp. 739–747.

[13] G. Entenman and S. W. Daniel, "A fully automatic hierachical compactor," in *Proceedings of the 22nd Design Automation Conference*, 1985, pp. 69–75.

[14] S. B. Akers, J. M. Geyer, and D. L. Roberts, "Mask layout with a single conductor layer," in *Proceedings of the 7th Design Automation Workshop*, 1970, pp. 7–16.

[15] F. Kato and H. Shiraishi, "Efficient compaction techniques for lsi layout," in *IEEE 1985 International Conference of Computer Design*, 1985, pp. 646–649.

[16] M. Ishikawa, T. Matsuda, and S. Goto, "Compaction based custom LSI layout design method," in *Proceedings of the International Conference on Computers-Aided Design*, 1985, pp. 346–348.

[17] A. E. Dunlop, "Slim - the translation of symbolic layout into mask data," in *Proceedings of the 17th Design Automation Conference*, 1980, pp. 595–602.

[18] W. L. Schiele, "Improved compaction by minimized length of wires," in *Proceedings of the 20th Design Automation Conference*, June 1983, pp. 121–127.

[19] Y. Z. Lio and C. K. Wong, "An algorithm to compact VLSI symbolic layout with mixed constraints," in *Proceedings of the 20th Design Automation Conference*, June 1983, pp. 107–112.

[20] M. D. Huang and K. Steiglitz, "A heirarchical compaction algorithm with low page-fault complexity," in *1984 Conference on Advanced Research in VLSI, M.I.T.*, January 1984, pp. 203–212.

[21] C. Kingsley, "A hierarchical, error tolerant compactor," in *Proceedings of the 21st Design Automation Conference*, June 1984, pp. 126–132.

[22] M. Hsueh and D. Pederson, "Computer-aided layout of LSI circuit building-blocks," in *Proceedings of International Symp. on Circuits and Systems*, 1979, pp. 474–477.

[23] P. Cook, "Constraint solver for generalized IC layout," *IBM Journal of Research and Development*, September 1984, pp. 581–589.

[24] T. Hedges, W. Dawson, and Y. Cho, "Bitmap graph algorithm for compaction," in *Proceedings of the International Conference on Computers-Aided Design*, 1985, pp. 340–342.

[25] R. C. Mosteller, "Rest: a leaf cell design system," in *VLSI 81*, (J. P. Gray, ed.), Academic Press, 1981, pp. 163–172.

[26] J. Do and W. M. Dawson, "Spacer ii: a well-behaved IC layout compactor," in *VLSI 85*, (E. Hörbst, ed.), North-Holland, 1985, pp. 283–291.

[27] R. D. Fiebrich, Y. Z. Lioa, G. Koppelman, and E. Adams, "Psi: a symbolic layout system," *IBM Journal of Research and Development*, September 1984, pp. 572–580.

[28] S. Rubin, *Computer Aids for VLSI Design.* Addison-Wesley, 1987.

[29] J. Ullman, *Computational Aspects of VLSI.* Computer Science Press, 1984.

[30] J. D. Williams, "STICKS–a graphical compiler for high level LSI designs," in *Proceedings AFIPS Conference 47*, June 1978, pp. 289–295.

[31] G. Kedem and H. Watanabe, "Graph optimization techniques for IC layout and compaction," in *Proceedings of the 20th Design Automation Conference*, June 1983, pp. 113–120.

[32] M. Schlag, Y. Z. Lioa, and C. Wong, "An algortihm for optimal two-dimensional compaction of VLSI layouts," in *Proceedings of the International Conference on Computers-Aided Design*, 1983, pp. 88–89.

[33] G. Kedem and H. Watanabe, "Graph optimization techniques for IC layout and compaction," *IEEE Transaction of Computer Aided Design*, vol. CAD-3, no. 1, October 1982, pp. 12–19.

[34] W. Wolf, R. Mathews, J. Newkirk, and R. Dutton, "Two-dimensional compaction strategies," in *Proceedings of the International Conference on Computers-Aided Design*, 1983, pp. 90–91.

[35] T. Lengauer, "The complexity of compacting heirarchically specified layouts of integrated circuits," in *Proceedings of the IEEE Conference*, 1982, pp. 358–368.

[36] S. Sahni and R. Kane, "A systolic design rule checker," in *Proceedings of the 21st Design Automation Conference*, 1984, pp. 243–250.

[37] S. Sechrest, *An Introductory 4.3BSD Interprocess Communication Tutorial.* Computer Science Research Group, Computer Science Division, University of California, Berkeley, 1986.

[38] S. J. Leffler, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek, *An Advanced 4.3BSD Interprocess Communication Tutorial.* Computer Science Research Group, Computer Science Division, University of California, Berkeley, 1986.

[39] *Remote Procedure Call Programming Guide.* Sun Microsystems, February 1986.

[40] *External Data Reresentations Protocol Specification.* Sun Microsystems, February 1986.

[41] *Remote Procedure Call Protocol Specification.* Sun Microsystems, February 1986.

[42] S. C. Johnson, *Yacc: Yet Anoterh Compiler-Compiler.* AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[43] E. Horowitz and S. Sahni, *Fundementals of Computer Algorithms.* Computer Science Press, 1984.

[44] J. Ousterhout, *Editing VLSI Circuits with Caesar.* Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, March 1983.

[45] C. Mead and L. Conway, *Introduction to VLSI System.* Addison-Wesley, 1980.

[46] R. Byrne, "Fred: an implementation of an arbitary 2-place quaternary function in cmos," in *1986 Canadian Conference on VLSI*, October 1986, pp. 145–150.

Appendix A

Layout Language Grammar

Layout Language Grammar

```
cell        : BEGINW ident EOL stmtlist END ident EOL
stmtlist    : stmt EOL
            | stmtlist stmt EOL
stmt        : INSTANCE ID or position
            | PIN ID ptype position net
            | WIRE wtype w_width path net well_id
            | DEVICE dtype position d_opt or snet gnet dnet well_id
            | CONTACT ctype position or net well_id
position    : '(' INT ',' INT ')'
path        : position position
            | path position
wtype       : ID
ptype       : ID
dtype       : ID
ctype       : ID
or          : OR '=' ID
d_opt       : d_width d_len
d_len       : /* empty */
            | LEN '=' INT
d_width     : /* empty */
            | WIDTH '=' INT
w_width     : /* empty */
            | WIDTH '=' INT
net         : NET '=' INT
            | /* empty */
snet        : SNET '=' INT
            | /* empty */
gnet        : GNET '=' INT
            | /* empty */
dnet        : DNET '=' INT
```

```
                | /* empty */
well_id         : WELL_ID '=' INT
                | /* empty */
ident           : ID
                | /* empty */
```

Appendix B

Timing Data for Arlene

Layout:nand_4 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	9.19	3.54 [39%]	4.47 [49%]	0.44 [5%]	53.4	1
1	7.64	3.08 [40%]	3.17 [41%]	0.57 [7%]	58.5	2
1	7.38	3.00 [41%]	3.39 [46%]	0.42 [6%]	63.9	3
2	12.89	6.00 [47%]	3.71 [29%]	1.16 [9%]	30.6	4
2	9.19	3.22 [35%]	3.64 [40%]	1.28 [14%]	52.0	5
2	8.95	2.96 [33%]	3.09 [35%]	0.96 [11%]	48.4	6
3	11.49	3.98 [35%]	3.57 [31%]	2.11 [18%]	40.8	7
3	11.53	4.11 [36%]	3.24 [28%]	2.21 [19%]	40.3	8
3	10.69	3.86 [36%]	3.76 [35%]	1.37 [13%]	46.6	9

Note	Processors [cpu usage]		
1	pooky [81.0%]		
2	irma [99.2%]		
3	pooky [90.7%]		
4	irma [99.6%]	lyman [99.0%]	
5	pooky [87.3%]	irma [97.9%]	
6	lyman [98.9%]	irma [100.0%]	
7	lyman [84.7%]	irma [78.1%]	pooky [68.2%]
8	irma [93.5%]	lyman [88.1%]	pooky [73.5%]
9	irma [99.6%]	pooky [75.4%]	lyman [43.2%]

Table B.1: Timing of nand_4 on client Arlene

Layout:nand_8 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	13.83	4.07 [29%]	7.83 [57%]	1.12 [8%]	58.3	1
1	15.04	6.34 [42%]	6.28 [42%]	0.66 [4%]	39.7	2
1	17.32	5.61 [32%]	9.16 [53%]	1.02 [6%]	40.5	3
2	12.68	3.80 [30%]	5.59 [44%]	1.70 [13%]	49.5	4
2	15.42	4.82 [31%]	7.28 [47%]	1.98 [13%]	44.0	5
2	23.49	11.58 [49%]	7.59 [32%]	1.88 [8%]	25.8	6
3	16.16	5.41 [33%]	6.19 [38%]	2.58 [16%]	42.6	7
3	17.58	6.13 [35%]	6.37 [36%]	2.58 [15%]	36.3	8
3	14.25	5.36 [38%]	4.80 [34%]	2.25 [16%]	47.2	9

Note	Processors [cpu usage]		
1	pooky [79.6%]		
2	irma [99.8%]		
3	pooky [76.2%]		
4	irma [99.6%]	lyman [80.2%]	
5	pooky [83.1%]	lyman [99.1%]	
6	irma [99.8%]	lyman [98.4%]	
7	lyman [92.5%]	irma [98.9%]	pooky [86.5%]
8	lyman [97.0%]	irma [99.7%]	pooky [88.6%]
9	irma [100.0%]	lyman [95.7%]	pooky [79.4%]

Table B.2: Timing of nand_8 on client Arlene

Layout:ff Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	50.48	13.34 [26%]	34.82 [69%]	1.81 [4%]	74.9	1
1	44.95	11.99 [27%]	30.61 [68%]	1.68 [4%]	80.9	2
1	45.87	11.82 [26%]	32.02 [70%]	1.51 [3%]	84.5	3
2	39.24	12.22 [31%]	21.92 [56%]	4.15 [11%]	74.6	4
2	42.75	15.57 [36%]	22.14 [52%]	3.96 [9%]	63.0	5
2	38.15	11.19 [29%]	22.70 [60%]	2.96 [8%]	81.6	6
3	39.12	13.66 [35%]	18.76 [48%]	5.61 [14%]	68.1	7
3	57.58	12.65 [22%]	37.80 [66%]	6.04 [10%]	71.0	8
3	46.16	14.57 [32%]	23.34 [51%]	7.24 [16%]	62.8	9

Note	Processors [cpu usage]		
1	lyman [85.1%]		
2	irma [93.5%]		
3	lyman [91.6%]		
4	irma [92.4%]	lyman [93.3%]	
5	lyman [93.5%]	irma [96.5%]	
6	irma [89.8%]	lyman [87.0%]	
7	irma [88.2%]	lyman [90.3%]	pooky [94.4%]
8	irma [41.8%]	lyman [90.3%]	pooky [84.1%]
9	irma [69.2%]	lyman [88.3%]	pooky [81.5%]

Table B.3: Timing of ff on client Arlene

Layout:tt2 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	47.73	12.03 [25%]	33.38 [70%]	1.68 [4%]	82.8	1
1	52.32	16.35 [31%]	33.67 [64%]	1.75 [3%]	66.5	2
1	44.76	11.00 [25%]	31.55 [71%]	1.66 [4%]	87.2	3
2	37.33	10.93 [29%]	22.59 [61%]	3.02 [8%]	89.1	4
2	38.92	12.52 [32%]	22.61 [58%]	3.01 [8%]	82.1	5
2	43.94	13.65 [31%]	26.52 [60%]	3.04 [7%]	76.7	6
3	47.05	11.67 [25%]	28.48 [61%]	5.83 [12%]	78.2	7
3	48.19	10.76 [22%]	29.14 [60%]	7.23 [15%]	75.1	8
3	52.80	11.17 [21%]	34.80 [66%]	5.78 [11%]	81.7	9

Note	Processors [cpu usage]		
1	lyman [92.6%]		
2	irma [90.2%]		
3	lyman [97.3%]		
4	irma [89.1%]	lyman [89.9%]	
5	irma [88.7%]	lyman [87.2%]	
6	irma [76.7%]	lyman [93.4%]	
7	irma [60.0%]	lyman [93.3%]	pooky [78.7%]
8	lyman [76.0%]	irma [66.7%]	pooky [72.0%]
9	irma [44.8%]	lyman [82.4%]	pooky [76.3%]

Table B.4: Timing of tt2 on client Arlene

Layout:tt3 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	88.37	16.90 [19%]	68.27 [77%]	2.47 [3%]	89.4	1
1	84.26	16.80 [20%]	64.47 [77%]	2.45 [3%]	92.5	2
1	95.13	21.78 [23%]	70.03 [74%]	2.81 [3%]	74.4	3
2	65.13	16.97 [26%]	42.94 [66%]	4.42 [7%]	91.5	4
2	70.52	21.41 [30%]	42.96 [61%]	5.23 [7%]	74.0	5
2	71.55	24.96 [35%]	41.20 [58%]	4.58 [6%]	68.7	6
3	78.72	19.90 [25%]	47.56 [60%]	8.87 [11%]	72.7	7
3	73.28	22.63 [31%]	40.29 [55%]	9.33 [13%]	68.4	8
3	71.30	18.32 [26%]	42.58 [60%]	9.36 [13%]	77.9	9

Note	Processors [cpu usage]		
1	lyman [94.4%]		
2	irma [98.6%]		
3	lyman [92.1%]		
4	irma [96.3%]	lyman [89.8%]	
5	irma [96.4%]	lyman [87.2%]	
6	irma [99.8%]	lyman [86.2%]	
7	irma [88.2%]	lyman [89.3%]	pooky [79.7%]
8	irma [89.8%]	lyman [91.0%]	pooky [92.7%]
9	irma [86.0%]	lyman [90.1%]	pooky [88.1%]

Table B.5: Timing of tt3 on client Arlene

Layout:tt4 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	125.94	25.39 [20%]	92.51 [73%]	7.36 [6%]	79.8	1
1	136.10	26.38 [19%]	105.15 [77%]	3.96 [3%]	85.9	2
1	123.05	28.09 [23%]	90.87 [74%]	3.53 [3%]	83.8	3
2	123.01	42.14 [34%]	66.34 [54%]	12.99 [11%]	54.2	4
2	152.49	30.18 [20%]	111.51 [73%]	9.79 [6%]	73.6	5
2	99.92	29.55 [30%]	62.75 [63%]	6.81 [7%]	81.0	6
3	99.80	24.84 [25%]	61.70 [62%]	12.11 [12%]	82.2	7
3	108.36	26.93 [25%]	67.14 [62%]	13.30 [12%]	75.3	8
3	222.45	23.62 [11%]	95.74 [43%]	102.12 [46%]	24.1	9

Note	Processors [cpu usage]		
1	irma [97.5%]		
2	lyman [87.9%]		
3	irma [99.0%]		
4	lyman [89.0%]	irma [94.2%]	
5	irma [87.3%]	pooky [86.0%]	
6	lyman [93.0%]	irma [88.7%]	
7	irma [99.3%]	lyman [95.4%]	pooky [85.4%]
8	lyman [91.8%]	irma [92.5%]	pooky [79.9%]
9	irma [99.6%]	lyman [91.3%]	pooky [55.3%]

Table B.6: Timing of tt4 on client Arlene

Layout:tt5 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	174.04	37.04 [21%]	131.93 [76%]	4.53 [3%]	82.9	1
1	206.02	36.80 [18%]	162.62 [79%]	6.03 [3%]	78.5	2
1	181.40	40.44 [22%]	134.06 [74%]	6.25 [3%]	74.2	3
2	127.63	35.33 [28%]	83.86 [66%]	7.73 [6%]	89.3	4
2	176.88	35.46 [20%]	128.59 [73%]	12.00 [7%]	79.0	5
2	146.77	41.13 [28%]	91.98 [63%]	12.80 [9%]	73.5	6
3	128.77	39.25 [30%]	72.55 [56%]	15.89 [12%]	76.2	7
3	132.12	31.62 [24%]	83.00 [63%]	16.29 [12%]	81.0	8
3	136.20	39.33 [29%]	78.47 [58%]	17.25 [13%]	74.7	9

Note	Processors [cpu usage]		
1	lyman [93.9%]		
2	irma [75.8%]		
3	lyman [92.4%]		
4	lyman [91.9%]	irma [96.8%]	
5	lyman [93.1%]	pooky [97.2%]	
6	irma [82.7%]	lyman [88.3%]	
7	irma [96.3%]	lyman [93.5%]	pooky [98.2%]
8	irma [68.5%]	lyman [94.9%]	pooky [90.4%]
9	lyman [86.1%]	irma [83.5%]	pooky [91.9%]

Table B.7: Timing of tt5 on client Arlene

Layout:tt6 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	378.74	39.20 [10%]	327.36 [86%]	11.65 [3%]	83.4	1
1	222.46	40.26 [18%]	176.28 [79%]	5.41 [2%]	94.3	2
1	294.41	40.78 [14%]	247.33 [84%]	5.75 [2%]	91.6	3
2	205.10	50.06 [24%]	132.55 [65%]	21.68 [11%]	69.8	4
2	316.26	46.44 [15%]	250.02 [79%]	18.96 [6%]	76.3	5
2	172.47	41.85 [24%]	118.70 [69%]	11.12 [6%]	91.1	6
3	170.77	53.55 [31%]	93.99 [55%]	21.52 [13%]	70.5	7
3	169.44	52.25 [31%]	93.48 [55%]	22.62 [13%]	71.3	8
3	219.80	42.77 [19%]	148.65 [68%]	27.26 [12%]	76.2	9

Note	Processors [cpu usage]		
1	irma [48.4%]		
2	lyman [91.5%]		
3	irma [64.7%]		
4	irma [73.1%]	lyman [91.4%]	
5	irma [39.2%]	lyman [89.8%]	
6	irma [81.0%]	lyman [91.5%]	
7	irma [99.1%]	lyman [98.0%]	pooky [98.2%]
8	irma [99.5%]	lyman [98.9%]	pooky [98.4%]
9	irma [50.2%]	lyman [91.8%]	pooky [97.2%]

Table B.8: Timing of tt6 on client Arlene

Layout:tt10 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	422.51	83.30 [20%]	327.41 [77%]	11.25 [3%]	95.2	1
1	404.05	87.07 [22%]	305.37 [76%]	11.07 [3%]	92.5	2
1	427.92	83.24 [19%]	331.94 [78%]	12.21 [3%]	94.6	3
2	299.67	87.39 [29%]	191.13 [64%]	20.41 [7%]	92.2	4
2	280.87	82.90 [30%]	176.91 [63%]	20.28 [7%]	95.4	5
2	288.05	83.76 [29%]	183.21 [64%]	20.27 [7%]	95.0	6
3	355.77	82.97 [23%]	232.72 [65%]	39.06 [11%]	87.2	7
3	335.72	83.95 [25%]	210.75 [63%]	39.99 [12%]	86.8	8
3	342.60	83.46 [24%]	215.96 [63%]	42.18 [12%]	84.6	9

Note	Processors [cpu usage]		
1	lyman [94.4%]		
2	irma [99.5%]		
3	lyman [93.0%]		
4	irma [90.7%]	lyman [93.4%]	
5	lyman [98.2%]	irma [99.7%]	
6	irma [94.3%]	lyman [89.9%]	
7	irma [98.1%]	lyman [94.5%]	pooky [91.6%]
8	lyman [91.2%]	irma [99.3%]	pooky [99.2%]
9	lyman [96.3%]	irma [99.7%]	pooky [97.3%]

Table B.9: Timing of tt10 on client Arlene

Layout:tt20 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	1648.69	539.70 [33%]	1045.16 [63%]	62.65 [4%]	53.7	1
1	1728.85	605.31 [35%]	1063.28 [62%]	59.26 [3%]	52.8	2
1	1406.00	323.53 [23%]	1046.17 [74%]	35.72 [3%]	84.6	3
2	901.70	268.20 [30%]	568.58 [63%]	64.14 [7%]	95.7	4
2	901.48	268.26 [30%]	568.19 [63%]	64.25 [7%]	95.8	5
2	899.54	273.36 [30%]	561.51 [62%]	63.90 [7%]	95.5	6
3	1143.94	268.97 [24%]	747.52 [65%]	126.33 [11%]	87.1	7
3	1352.63	270.06 [20%]	914.37 [68%]	167.19 [12%]	78.5	8
3	1171.02	268.58 [23%]	768.08 [66%]	133.05 [11%]	86.5	9

Note	Processors [cpu usage]		
1	irma [99.7%]		
2	lyman [99.7%]		
3	irma [99.6%]		
4	lyman [99.6%]	irma [99.7%]	
5	lyman [99.6%]	irma [99.6%]	
6	irma [99.7%]	lyman [99.7%]	
7	lyman [99.6%]	irma [99.6%]	pooky [98.9%]
8	irma [99.7%]	lyman [93.8%]	pooky [82.1%]
9	irma [99.7%]	lyman [90.0%]	pooky [96.7%]

Table B.10: Timing of tt20 on client Arlene

Layout:shift_30 Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	1256.38	482.75 [38%]	742.20 [59%]	30.60 [2%]	56.5	1
1	1685.87	809.97 [48%]	804.81 [48%]	70.04 [4%]	32.3	2
1	1666.59	527.00 [32%]	1089.29 [65%]	49.70 [3%]	49.6	3
2	1404.20	710.40 [51%]	521.18 [37%]	170.97 [12%]	38.0	4
2	1255.12	539.82 [43%]	633.49 [50%]	80.17 [6%]	50.7	5
2	1535.37	552.67 [36%]	876.61 [57%]	104.94 [7%]	47.4	6
3	1407.93	844.05 [60%]	405.59 [29%]	155.60 [11%]	35.0	7
3	1203.63	605.80 [50%]	439.12 [36%]	156.05 [13%]	44.8	8
3	1153.77	502.20 [44%]	466.82 [40%]	183.31 [16%]	49.9	9

Note	Processors [cpu usage]		
1	irma [99.6%]		
2	irma [92.5%]		
3	lyman [70.0%]		
4	irma [80.2%]	pooky [82.8%]	
5	lyman [64.9%]	irma [98.3%]	
6	irma [93.7%]	pooky [96.6%]	
7	lyman [71.6%]	irma [86.9%]	pooky [77.0%]
8	lyman [65.8%]	irma [99.7%]	pooky [90.0%]
9	irma [78.8%]	lyman [62.6%]	pooky [89.5%]

Table B.11: Timing of shift_30 on client Arlene

Layout:bit Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	4169.94	1913.24 [46%]	2071.95 [50%]	183.40 [4%]	37.4	1
1	4902.04	2578.63 [53%]	2130.97 [43%]	190.50 [4%]	29.0	2
1	6228.10	2413.05 [39%]	3598.15 [58%]	215.43 [3%]	31.0	3
2	4888.64	2090.80 [43%]	2415.58 [49%]	380.56 [8%]	33.8	4
2	4731.32	2357.16 [50%]	1821.27 [38%]	550.52 [12%]	29.5	5
2	4382.36	1873.10 [43%]	2136.76 [49%]	370.96 [8%]	36.9	6

Note	Processors [cpu usage]	
1	irma [97.6%]	
2	irma [95.3%]	
3	lyman [57.5%]	
4	lyman [45.2%]	irma [99.6%]
5	irma [99.6%]	lyman [56.7%]
6	irma [99.4%]	lyman [48.6%]

Table B.12: Timing of bit on client Arlene

Layout:mem Client: Arlene						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	7232.80	2077.90 [29%]	4854.52 [67%]	299.02 [4%]	43.4	1
2	5970.61	2679.32 [45%]	2801.93 [47%]	487.24 [8%]	34.0	2
2	5480.80	2265.41 [41%]	2670.41 [49%]	543.22 [10%]	38.1	3
2	5361.69	2261.66 [42%]	2686.91 [50%]	411.83 [8%]	39.5	4

Note	Processors [cpu usage]	
1	lyman [51.7%]	
2	lyman [46.4%]	irma [99.7%]
3	lyman [48.7%]	irma [99.7%]
4	lyman [48.4%]	irma [99.6%]

Table B.13: Timing of mem on client Arlene

Appendix C

Timing Data for Jon

Layout:nand_4 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	4.00	0.76 [19%]	2.86 [72%]	0.22 [5%]	47.5	1
1	4.00	0.78 [20%]	2.86 [72%]	0.20 [5%]	55.9	2
1	5.08	1.76 [35%]	2.98 [59%]	0.22 [4%]	25.9	3
2	4.28	0.80 [19%]	2.80 [65%]	0.42 [10%]	39.7	4
2	4.36	0.76 [17%]	2.98 [68%]	0.38 [9%]	57.5	5
2	4.22	0.80 [19%]	2.80 [66%]	0.36 [9%]	52.7	6
3	4.28	0.78 [18%]	2.62 [61%]	0.54 [13%]	44.8	7
3	4.96	0.80 [16%]	3.20 [65%]	0.56 [11%]	46.7	8
3	5.16	1.60 [31%]	2.62 [51%]	0.58 [11%]	29.0	9

Note	Processors [cpu usage]		
1	irma [99.6%]		
2	pooky [100.0%]		
3	lyman [97.9%]		
4	irma [99.6%]	pooky [82.5%]	
5	lyman [96.5%]	pooky [98.0%]	
6	irma [99.3%]	pooky [99.0%]	
7	irma [99.6%]	lyman [99.3%]	pooky [93.8%]
8	irma [99.6%]	pooky [60.1%]	lyman [100.0%]
9	pooky [99.6%]	lyman [97.3%]	irma [93.9%]

Table C.1: Timing of nand_4 on client jon

Layout:nand_8 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	7.18	1.02 [14%]	5.72 [80%]	0.28 [4%]	53.3	1
1	7.24	1.02 [14%]	5.72 [79%]	0.30 [4%]	52.6	2
1	7.34	1.02 [14%]	5.88 [80%]	0.28 [4%]	56.6	3
2	8.30	1.00 [12%]	6.46 [78%]	0.52 [6%]	50.5	4
2	7.26	1.00 [14%]	5.50 [76%]	0.54 [7%]	50.5	5
2	7.52	1.00 [13%]	5.72 [76%]	0.56 [7%]	59.1	6
3	7.34	1.44 [20%]	4.76 [65%]	0.78 [11%]	40.6	7
3	6.74	1.04 [15%]	4.64 [69%]	0.76 [11%]	49.5	8
3	6.96	1.20 [17%]	4.66 [67%]	0.78 [11%]	45.8	9

Note	Processors [cpu usage]		
1	pooky [99.8%]		
2	irma [99.8%]		
3	lyman [98.1%]		
4	pooky [81.9%]	irma [100.0%]	
5	lyman [96.3%]	irma [99.6%]	
6	pooky [92.3%]	irma [99.6%]	
7	irma [97.4%]	lyman [98.6%]	pooky [98.6%]
8	pooky [99.3%]	irma [98.6%]	lyman [97.9%]
9	lyman [99.3%]	pooky [100.0%]	irma [99.3%]

Table C.2: Timing of nand_8 on client jon

Layout:ff Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	33.28	3.68 [11%]	28.30 [85%]	1.04 [3%]	58.4	1
1	33.66	3.90 [12%]	28.46 [85%]	1.02 [3%]	55.6	2
1	33.10	3.54 [11%]	28.24 [85%]	1.04 [3%]	61.0	3
2	27.76	3.68 [13%]	21.42 [77%]	2.40 [9%]	49.5	4
2	25.78	3.58 [14%]	20.06 [78%]	1.84 [7%]	54.5	5
2	25.92	3.52 [14%]	20.18 [78%]	1.84 [7%]	55.2	6
3	22.06	3.72 [17%]	15.20 [69%]	2.66 [12%]	52.2	7
3	21.76	3.58 [16%]	15.24 [70%]	2.60 [12%]	53.0	8
3	23.28	3.84 [16%]	16.40 [70%]	2.64 [11%]	50.4	9

Note	Processors [cpu usage]		
1	irma [99.7%]		
2	lyman [99.6%]		
3	pooky [99.6%]		
4	pooky [93.8%]	lyman [99.5%]	
5	irma [99.8%]	pooky [96.1%]	
6	lyman [99.7%]	irma [99.8%]	
7	irma [99.7%]	lyman [99.7%]	pooky [93.6%]
8	pooky [100.0%]	irma [99.6%]	lyman [99.6%]
9	pooky [92.6%]	lyman [99.5%]	irma [99.1%]

Table C.3: Timing of ff on client jon

Layout:tt2 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	35.52	3.96 [11%]	30.08 [85%]	1.28 [4%]	56.2	1
1	35.68	3.84 [11%]	30.28 [85%]	1.18 [3%]	56.2	2
1	34.96	3.32 [9%]	30.36 [87%]	1.18 [3%]	64.0	3
2	26.80	4.34 [16%]	20.14 [75%]	2.08 [8%]	48.7	4
2	28.42	5.54 [19%]	20.08 [71%]	2.20 [8%]	42.1	5
2	26.70	3.92 [15%]	20.12 [75%]	2.16 [8%]	50.4	6
3	22.98	3.84 [17%]	15.78 [69%]	2.96 [13%]	49.9	7
3	22.84	3.90 [17%]	15.32 [67%]	2.96 [13%]	48.2	8
3	24.14	4.54 [19%]	16.20 [67%]	3.00 [12%]	46.8	9

Note	Processors [cpu usage]		
1	pooky [99.6%]		
2	irma [99.4%]		
3	lyman [99.6%]		
4	irma [99.1%]	lyman [100.0%]	
5	pooky [99.3%]	lyman [99.8%]	
6	irma [99.4%]	pooky [92.5%]	
7	irma [97.7%]	pooky [97.5%]	lyman [95.9%]
8	pooky [99.4%]	irma [99.8%]	lyman [99.9%]
9	lyman [99.7%]	pooky [92.1%]	irma [99.2%]

Table C.4: Timing of tt2 on client jon

Layout:tt3 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	74.16	7.56 [10%]	64.44 [87%]	1.82 [2%]	50.0	1
1	73.28	7.20 [10%]	64.08 [87%]	1.84 [3%]	50.3	2
1	76.82	7.34 [10%]	67.26 [88%]	1.84 [2%]	50.9	3
2	52.10	6.94 [13%]	41.48 [80%]	3.32 [6%]	48.6	4
2	52.32	7.18 [14%]	41.58 [79%]	3.26 [6%]	47.7	5
2	52.04	7.04 [14%]	41.50 [80%]	3.26 [6%]	48.1	6
3	42.56	7.10 [17%]	30.18 [71%]	4.68 [11%]	44.6	7
3	42.66	7.08 [17%]	30.52 [72%]	4.62 [11%]	46.8	8
3	42.92	7.56 [18%]	30.14 [70%]	4.62 [11%]	45.7	9

Note	Processors [cpu usage]		
1	lyman [99.5%]		
2	irma [99.7%]		
3	pooky [95.4%]		
4	lyman [99.9%]	irma [99.7%]	
5	pooky [99.2%]	irma [99.5%]	
6	lyman [99.8%]	irma [99.8%]	
7	lyman [99.8%]	irma [99.7%]	pooky [99.5%]
8	irma [99.8%]	pooky [95.8%]	lyman [99.9%]
9	lyman [99.7%]	irma [99.6%]	pooky [99.2%]

Table C.5: Timing of tt3 on client jon

Layout:tt4 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	116.28	10.72 [9%]	102.78 [88%]	2.60 [2%]	49.3	1
1	108.02	11.46 [11%]	93.72 [87%]	2.58 [2%]	47.0	2
1	110.34	10.70 [10%]	96.40 [87%]	3.02 [3%]	46.5	3
2	79.32	10.78 [14%]	63.56 [80%]	4.50 [6%]	46.7	4
2	73.42	11.24 [15%]	57.32 [78%]	4.64 [6%]	45.3	5
2	76.22	10.94 [14%]	59.18 [78%]	5.86 [8%]	42.2	6
3	60.80	10.64 [18%]	43.06 [71%]	6.50 [11%]	44.0	7
3	60.58	10.76 [18%]	42.70 [70%]	6.42 [11%]	44.8	8
3	61.18	11.22 [18%]	43.02 [70%]	6.64 [11%]	42.4	9

Note	Processors [cpu usage]		
1	lyman [89.1%]		
2	pooky [96.5%]		
3	irma [94.6%]		
4	pooky [90.9%]	lyman [94.2%]	
5	irma [99.7%]	pooky [94.2%]	
6	lyman [97.9%]	pooky [96.6%]	
7	pooky [96.2%]	lyman [99.7%]	irma [99.8%]
8	pooky [96.9%]	irma [99.8%]	lyman [99.3%]
9	pooky [96.6%]	irma [99.7%]	lyman [98.2%]

Table C.6: Timing of tt4 on client jon

Layout:tt5 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	145.88	14.78 [10%]	127.42 [87%]	3.54 [2%]	47.7	1
1	144.68	14.72 [10%]	126.06 [87%]	3.68 [3%]	47.3	2
1	146.24	15.24 [10%]	127.08 [87%]	3.74 [3%]	47.3	3
2	100.08	14.78 [15%]	78.42 [78%]	6.54 [7%]	44.8	4
2	100.94	14.98 [15%]	79.36 [79%]	6.26 [6%]	44.1	5
2	118.62	15.10 [13%]	91.96 [78%]	11.28 [10%]	36.4	6
3	91.02	14.66 [16%]	66.00 [73%]	10.02 [11%]	41.0	7
3	80.64	14.60 [18%]	56.74 [70%]	8.90 [11%]	42.8	8
3	83.28	15.08 [18%]	59.06 [71%]	8.78 [11%]	43.4	9

Note	Processors [cpu usage]		
1	pooky [95.4%]		
2	lyman [97.4%]		
3	pooky [95.7%]		
4	lyman [98.6%]	irma [87.1%]	
5	pooky [95.6%]	irma [90.3%]	
6	lyman [87.2%]	irma [73.6%]	
7	pooky [94.5%]	irma [79.1%]	lyman [98.9%]
8	lyman [98.0%]	pooky [97.1%]	irma [87.7%]
9	irma [89.9%]	pooky [97.3%]	lyman [97.8%]

Table C.7: Timing of tt5 on client jon

Layout:tt6 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	181.76	19.62 [11%]	157.40 [87%]	4.54 [2%]	48.0	1
1	184.70	19.44 [11%]	160.58 [87%]	4.52 [2%]	47.0	2
1	182.82	20.06 [11%]	157.92 [86%]	4.48 [2%]	46.6	3
2	128.50	19.88 [15%]	100.06 [78%]	8.16 [6%]	44.9	4
2	122.96	19.66 [16%]	95.14 [77%]	7.78 [6%]	44.2	5
2	127.06	19.82 [16%]	98.56 [78%]	8.16 [6%]	43.6	6
3	101.66	20.24 [20%]	69.52 [68%]	11.28 [11%]	41.2	7
3	101.60	19.72 [19%]	69.84 [69%]	11.68 [11%]	41.8	8
3	151.86	19.74 [13%]	119.64 [79%]	12.00 [8%]	41.8	9

Note	Processors [cpu usage]		
1	irma [99.6%]		
2	lyman [98.9%]		
3	irma [99.3%]		
4	pooky [95.2%]	lyman [93.6%]	
5	irma [99.8%]	pooky [97.9%]	
6	lyman [97.5%]	pooky [96.9%]	
7	irma [99.7%]	pooky [98.1%]	lyman [98.7%]
8	pooky [97.4%]	irma [99.1%]	lyman [95.5%]
9	irma [57.7%]	lyman [94.7%]	pooky [94.7%]

Table C.8: Timing of tt6 on client jon

Layout:tt10 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	365.22	44.26 [12%]	311.30 [85%]	9.42 [3%]	45.5	1
1	403.30	43.54 [11%]	349.78 [87%]	9.72 [2%]	45.3	2
1	358.60	43.78 [12%]	305.20 [85%]	9.44 [3%]	46.0	3
2	239.58	43.68 [18%]	177.08 [74%]	18.32 [8%]	41.5	4
2	234.48	44.10 [19%]	172.40 [74%]	17.76 [8%]	42.8	5
2	239.32	44.64 [19%]	176.64 [74%]	17.46 [7%]	41.7	6
3	206.60	43.68 [21%]	136.92 [66%]	25.40 [12%]	40.1	7
3	191.80	43.90 [23%]	123.42 [64%]	24.08 [13%]	41.4	8
3	206.60	42.96 [21%]	139.42 [67%]	23.80 [12%]	42.1	9

Note	Processors [cpu usage]		
1	pooky [97.4%]		
2	lyman [89.0%]		
3	irma [99.6%]		
4	pooky [96.9%]	lyman [91.9%]	
5	irma [99.7%]	pooky [97.7%]	
6	lyman [97.9%]	pooky [97.2%]	
7	lyman [90.8%]	irma [99.6%]	pooky [97.6%]
8	pooky [97.3%]	irma [99.7%]	lyman [94.6%]
9	lyman [88.1%]	irma [99.7%]	pooky [95.8%]

Table C.9: Timing of tt10 on client jon

Layout:tt20 Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	1257.54	147.28 [12%]	1078.74 [86%]	31.26 [2%]	44.0	1
1	1272.26	146.44 [12%]	1094.56 [86%]	31.08 [2%]	44.6	2
1	1257.98	146.74 [12%]	1079.92 [86%]	30.90 [2%]	44.4	3
2	773.14	146.66 [19%]	570.80 [74%]	55.14 [7%]	41.6	4
2	793.40	146.62 [18%]	590.96 [74%]	55.44 [7%]	41.7	5
2	806.60	147.82 [18%]	601.90 [75%]	56.52 [7%]	41.4	6
3	617.92	146.74 [24%]	392.20 [63%]	78.52 [13%]	39.8	7
3	615.38	146.52 [24%]	387.92 [63%]	80.34 [13%]	39.3	8
3	667.64	147.42 [22%]	441.40 [66%]	78.50 [12%]	40.3	9

Note	Processors [cpu usage]		
1	pooky [96.6%]		
2	lyman [96.6%]		
3	pooky [96.5%]		
4	pooky [97.8%]	lyman [96.5%]	
5	irma [95.3%]	pooky [97.3%]	
6	lyman [94.2%]	irma [94.2%]	
7	pooky [97.6%]	irma [99.5%]	lyman [96.9%]
8	irma [99.7%]	pooky [97.8%]	lyman [96.4%]
9	lyman [90.3%]	irma [99.7%]	pooky [97.6%]

Table C.10: Timing of tt20 on client jon

Layout:bit Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	2525.30	394.22 [16%]	2065.30 [82%]	63.40 [3%]	45.0	1
1	2925.96	395.76 [14%]	2459.38 [84%]	69.70 [2%]	44.0	2
1	2733.70	395.84 [14%]	2259.48 [83%]	77.46 [3%]	43.6	3
2	1608.94	393.94 [24%]	1097.98 [68%]	115.14 [7%]	42.9	4
2	1676.52	393.92 [23%]	1166.78 [70%]	114.94 [7%]	42.7	5
2	1652.02	394.28 [24%]	1139.82 [69%]	116.74 [7%]	42.6	6
3	1321.28	392.02 [30%]	762.00 [58%]	163.82 [12%]	40.7	7
3	1304.22	394.10 [30%]	747.02 [57%]	161.18 [12%]	41.0	8
3	1330.00	390.88 [29%]	775.40 [58%]	161.88 [12%]	41.1	9

Note	Processors [cpu usage]		
1	lyman [98.4%]		
2	lyman [83.5%]		
3	pooky [90.6%]		
4	lyman [97.8%]	irma [98.7%]	
5	pooky [91.9%]	lyman [96.7%]	
6	irma [96.7%]	lyman [89.9%]	
7	lyman [96.9%]	irma [97.2%]	pooky [95.0%]
8	irma [99.6%]	pooky [97.8%]	lyman [96.9%]
9	irma [99.7%]	lyman [96.1%]	pooky [97.9%]

Table C.11: Timing of bit on client jon

Layout:mem Client: jon						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	1828.26	356.14 [19%]	1307.96 [72%]	161.40 [9%]	61.7	1
2	2017.52	284.66 [14%]	1505.88 [75%]	225.84 [11%]	56.2	2
3	4196.62	301.40 [7%]	3534.68 [84%]	358.90 [9%]	45.2	3
3	3005.08	341.84 [11%]	2346.64 [78%]	314.72 [10%]	48.3	4
3	6758.86	274.38 [4%]	6082.88 [90%]	400.68 [6%]	44.2	5

Note	Processors [cpu usage]		
1	lyman [99.5%]	irma [94.0%]	
2	irma [99.6%]	lyman [81.9%]	
3	pooky [28.5%]	lyman [99.1%]	irma [94.6%]
4	irma [89.0%]	pooky [37.5%]	lyman [89.4%]
5	irma [98.8%]	lyman [99.6%]	pooky [25.4%]

Table C.12: Timing of mem on client jon

Appendix D

Timing Data for Suna

Layout:nand_4 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	3.62	2.08 [57%]	1.12 [31%]	0.10 [3%]	28.3	1
1	3.54	2.16 [61%]	1.14 [32%]	0.10 [3%]	29.5	2
1	3.32	1.92 [58%]	1.12 [34%]	0.08 [2%]	38.7	3
2	3.46	1.94 [56%]	1.06 [31%]	0.20 [6%]	32.8	4
2	3.48	1.96 [56%]	1.06 [30%]	0.22 [6%]	37.4	5
2	3.60	2.08 [58%]	1.08 [30%]	0.22 [6%]	28.9	6
3	3.54	1.92 [54%]	1.00 [28%]	0.34 [10%]	36.9	7
3	3.74	2.12 [57%]	1.02 [27%]	0.26 [7%]	32.4	8
3	5.78	3.18 [55%]	1.66 [29%]	0.36 [6%]	27.7	9

Note	Processors [cpu usage]		
1	sunf [100.0%]		
2	sunc [100.0%]		
3	sunf [100.0%]		
4	sunc [100.0%]	sunb [100.0%]	
5	sunf [100.0%]	sunb [100.0%]	
6	sunc [100.0%]	sunb [100.0%]	
7	sunc [100.0%]	sunb [100.0%]	sunf [100.0%]
8	sunb [100.0%]	sunc [100.0%]	sunf [100.0%]
9	sunf [100.0%]	sunc [100.0%]	sunb [100.0%]

Table D.1: Timing of nand_4 on client suna

Layout:nand_8 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
1	4.70	2.18 [46%]	2.22 [47%]	0.14 [3%]	42.9	1
1	4.96	2.44 [49%]	2.20 [44%]	0.16 [3%]	39.3	2
1	4.68	2.16 [46%]	2.20 [47%]	0.14 [3%]	42.9	3
2	5.46	2.92 [53%]	2.06 [38%]	0.24 [4%]	35.5	4
2	4.68	2.18 [47%]	2.06 [44%]	0.22 [5%]	51.1	5
2	4.68	2.18 [47%]	2.02 [43%]	0.28 [6%]	37.8	6
3	6.14	2.38 [39%]	3.12 [51%]	0.38 [6%]	40.9	7
3	4.66	2.20 [47%]	1.78 [38%]	0.36 [8%]	43.5	8
3	5.12	2.60 [51%]	1.78 [35%]	0.34 [7%]	37.1	9

Note	Processors [cpu usage]		
1	sunb [100.0%]		
2	sunc [100.0%]		
3	sunf [100.0%]		
4	sunc [99.0%]	sunb [100.0%]	
5	sunf [100.0%]	sunc [100.0%]	
6	sunb [100.0%]	sunc [100.0%]	
7	sunb [100.0%]	sunc [62.3%]	sunf [100.0%]
8	sunf [100.0%]	sunb [100.0%]	sunc [100.0%]
9	sunb [100.0%]	sunc [100.0%]	sunf [100.0%]

Table D.2: Timing of nand_8 on client suna

Layout:ff Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	13.86	4.74 [34%]	7.76 [56%]	0.92 [7%]	56.7	1
2	13.86	4.92 [35%]	7.80 [56%]	0.90 [6%]	57.4	2
2	18.75	9.57 [51%]	8.13 [43%]	0.66 [4%]	33.7	3
3	14.54	4.50 [31%]	5.84 [40%]	2.94 [20%]	47.7	4
3	12.20	4.38 [36%]	6.04 [50%]	1.28 [10%]	61.7	5
3	12.66	4.96 [39%]	5.90 [47%]	1.30 [10%]	55.1	6
4	12.18	4.60 [38%]	5.90 [48%]	1.34 [11%]	57.0	7
4	13.54	5.44 [40%]	4.98 [37%]	2.18 [16%]	46.8	8
4	13.36	5.90 [44%]	5.12 [38%]	1.68 [13%]	49.9	9
5	13.64	4.62 [34%]	4.46 [33%]	3.92 [29%]	51.7	10
5	12.02	5.08 [42%]	4.32 [36%]	2.22 [18%]	56.2	11
5	12.70	5.68 [45%]	4.24 [33%]	2.36 [19%]	50.8	12
6	13.16	5.96 [45%]	3.94 [30%]	2.86 [22%]	51.2	13
6	16.12	4.96 [31%]	3.98 [25%]	5.88 [36%]	43.9	14
6	18.82	9.44 [50%]	4.00 [21%]	4.10 [22%]	39.6	15

Note	Processors [cpu usage]					
1	sunf [100.0%]	suna [100.0%]				
2	sunc [100.0%]	sunh [100.0%]				
3	suni [99.7%]	sunf [100.0%]				
4	sunb [100.0%]	sunc [100.0%]	sunh [100.0%]			
5	suni [98.6%]	sunf [100.0%]	suna [100.0%]			
6	sunc [100.0%]	sunh [100.0%]	sunb [100.0%]			
7	sunh [100.0%]	suni [100.0%]	suna [100.0%]	sunb [100.0%]		
8	sunf [100.0%]	sunc [100.0%]	sunb [100.4%]	suni [100.0%]		
9	suna [100.0%]	sunh [100.0%]	sunb [100.0%]	suni [100.0%]		
10	suni [100.0%]	sunf [100.0%]	sunc [100.0%]	sunh [100.6%]	sunb [100.0%]	
11	suna [100.0%]	sunc [100.0%]	sunf [100.0%]	sunh [100.0%]	sunb [100.0%]	
12	suni [100.0%]	sunc [100.0%]	sunf [100.0%]	sunh [100.0%]	sunb [65.5%]	
13	sunc [99.0%]	suna [100.0%]	suni [100.0%]	sunh [100.0%]	sunf [100.0%]	sunb [100.0%]
14	sunh [100.0%]	sunc [100.0%]	suni [100.0%]	sunf [100.0%]	suna [100.0%]	sunb [100.0%]
15	sunc [100.0%]	sunf [100.0%]	suna [100.0%]	sunh [99.4%]	sunb [100.0%]	suni [100.0%]

Table D.3: Timing of ff on client suna

Layout:tt2 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	13.92	4.94 [35%]	7.68 [55%]	0.90 [6%]	56.2	1
2	14.96	6.04 [40%]	7.72 [52%]	0.90 [6%]	48.6	2
2	15.06	6.04 [40%]	7.70 [51%]	1.06 [7%]	49.3	3
3	12.64	4.88 [39%]	5.88 [47%]	1.54 [12%]	56.8	4
3	13.22	5.24 [40%]	5.94 [45%]	1.54 [12%]	53.4	5
3	12.76	4.88 [38%]	5.86 [46%]	1.62 [13%]	54.1	6
4	12.90	5.48 [42%]	5.12 [40%]	1.76 [14%]	51.7	7
4	12.28	4.92 [40%]	4.74 [39%]	2.08 [17%]	55.5	8
4	11.98	4.60 [38%]	4.78 [40%]	2.08 [17%]	58.2	9
5	12.82	5.36 [42%]	4.42 [34%]	2.50 [20%]	52.1	10
5	12.38	5.12 [41%]	4.36 [35%]	2.38 [19%]	55.8	11
5	14.68	5.32 [36%]	6.28 [43%]	2.44 [17%]	56.5	12
6	13.76	4.94 [36%]	5.26 [38%]	2.86 [21%]	53.7	13
6	13.45	5.84 [43%]	3.87 [29%]	3.10 [23%]	49.6	14
6	17.72	8.66 [49%]	4.14 [23%]	3.56 [20%]	38.9	15

Note	Processors [cpu usage]					
1	sunb [100.0%]	sunf [98.8%]				
2	sunc [100.0%]	sunh [100.0%]				
3	sunf [100.0%]	suni [100.0%]				
4	sunf [100.0%]	suni [100.0%]	sunh [100.0%]			
5	suna [100.0%]	sunb [100.0%]	sunc [100.0%]			
6	sunc [99.7%]	sunf [100.0%]	suni [100.0%]			
7	suna [100.0%]	sunb [100.0%]	sunc [100.0%]	sunh [100.0%]		
8	sunf [100.0%]	suni [100.0%]	sunc [100.0%]	suna [100.0%]		
9	sunh [100.0%]	sunc [100.0%]	sunf [100.0%]	sunb [100.0%]		
10	suni [100.0%]	sunh [100.0%]	sunf [100.0%]	sunc [100.0%]	sunb [100.0%]	
11	sunh [100.0%]	sunf [100.0%]	sunb [100.0%]	suna [100.0%]	sunc [100.0%]	
12	sunb [78.5%]	sunc [100.0%]	sunf [100.0%]	suni [100.0%]	suna [100.0%]	
13	suna [98.5%]	sunc [80.9%]	sunb [97.9%]	sunh [98.0%]	sunf [100.0%]	suni [100.0%]
14	sunb [100.0%]	sunc [100.0%]	suna [100.5%]	sunh [100.0%]	sunf [100.0%]	suni [107.1%]
15	sunb [100.0%]	sunf [100.0%]	sunc [100.0%]	sunh [100.0%]	suni [100.0%]	suna [100.0%]

Table D.4: Timing of tt2 on client suna

Layout:tt3 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	26.02	8.26 [32%]	15.88 [61%]	1.56 [6%]	54.1	1
2	26.24	8.28 [32%]	15.98 [61%]	1.64 [6%]	53.6	2
2	26.61	8.04 [30%]	16.62 [62%]	1.67 [6%]	55.2	3
3	22.04	8.32 [38%]	11.54 [52%]	1.66 [8%]	57.6	4
3	23.40	8.84 [38%]	11.52 [49%]	2.68 [11%]	53.1	5
3	22.26	8.00 [36%]	11.68 [52%]	2.26 [10%]	54.1	6
4	21.22	7.88 [37%]	9.64 [45%]	3.34 [16%]	55.8	7
4	21.62	8.62 [40%]	9.80 [45%]	2.74 [13%]	51.3	8
4	24.06	9.76 [41%]	9.56 [40%]	3.40 [14%]	47.9	9
5	20.12	7.48 [37%]	8.04 [40%]	3.84 [19%]	56.2	10
5	21.06	7.88 [37%]	8.14 [39%]	4.38 [21%]	53.2	11
5	22.22	7.98 [36%]	9.94 [45%]	3.92 [18%]	55.8	12
6	23.26	10.58 [45%]	7.28 [31%]	4.92 [21%]	47.3	13
6	23.14	10.00 [43%]	7.74 [33%]	4.92 [21%]	47.5	14
6	21.86	8.58 [39%]	8.32 [38%]	4.26 [19%]	52.6	15

Note	Processors [cpu usage]					
1	sunc [100.0%]	sunh [100.0%]				
2	sunb [100.0%]	sunf [100.0%]				
3	suni [100.0%]	sunh [100.0%]				
4	sunb [100.0%]	sunc [100.0%]	suna [100.0%]			
5	suni [100.0%]	sunh [100.0%]	sunf [100.0%]			
6	sunc [100.0%]	sunb [100.0%]	suna [100.0%]			
7	suni [98.4%]	suna [99.1%]	sunf [100.0%]	sunh [100.0%]		
8	sunc [100.0%]	sunf [100.0%]	suna [100.0%]	sunh [100.0%]		
9	sunb [100.0%]	suni [100.0%]	suna [100.0%]	sunf [100.0%]		
10	suna [99.5%]	sunb [100.0%]	suni [100.0%]	sunh [100.0%]	sunf [100.0%]	
11	sunf [100.0%]	sunb [100.0%]	sunh [99.0%]	suni [100.0%]	sunc [100.0%]	
12	suna [100.0%]	suni [100.0%]	sunc [100.0%]	sunh [83.7%]	sunb [100.0%]	
13	suni [100.0%]	sunf [100.3%]	sunh [99.7%]	sunb [100.0%]	sunc [100.0%]	suna [100.0%]
14	sunc [100.0%]	sunb [99.1%]	suna [100.0%]	suni [95.5%]	sunf [99.3%]	sunh [100.0%]
15	sunh [99.7%]	sunb [100.0%]	sunf [90.5%]	suni [100.0%]	suna [100.0%]	sunc [99.0%]

Table D.5: Timing of tt3 on client suna

Layout:tt4 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	37.52	13.06 [35%]	21.82 [58%]	2.30 [6%]	50.3	1
2	37.18	12.56 [34%]	21.96 [59%]	2.28 [6%]	50.9	2
2	37.00	12.66 [34%]	21.96 [59%]	2.02 [5%]	52.2	3
3	38.06	12.82 [34%]	20.30 [53%]	4.28 [11%]	51.0	4
3	39.04	17.10 [44%]	17.84 [46%]	3.82 [10%]	43.2	5
3	30.52	12.04 [39%]	15.94 [52%]	2.24 [7%]	57.4	6
4	30.48	13.20 [43%]	12.38 [41%]	4.58 [15%]	50.3	7
4	29.86	12.26 [41%]	12.71 [43%]	4.36 [15%]	51.5	8
4	35.24	17.70 [50%]	12.54 [36%]	4.56 [13%]	43.2	9
5	31.60	14.26 [45%]	10.76 [34%]	6.02 [19%]	48.0	10
5	28.56	11.80 [41%]	10.82 [38%]	3.80 [13%]	53.3	11
5	28.68	11.38 [40%]	10.86 [38%]	5.88 [21%]	55.0	12
6	32.38	14.46 [45%]	9.98 [31%]	7.36 [23%]	47.8	13
6	29.90	13.86 [46%]	10.08 [34%]	5.44 [18%]	50.8	14
6	34.00	13.60 [40%]	12.74 [37%]	7.10 [21%]	49.2	15

Note	Processors [cpu usage]					
1	sunb [100.1%]	sunh [100.0%]				
2	suni [100.0%]	sunc [100.0%]				
3	sunb [100.1%]	sunf [100.0%]				
4	sunh [83.4%]	suna [100.0%]	suni [100.0%]			
5	sunb [92.4%]	sunc [100.0%]	sunf [100.0%]			
6	sunf [100.0%]	sunh [100.0%]	suni [100.0%]			
7	sunc [100.0%]	sunh [100.0%]	sunb [100.0%]	suni [100.0%]		
8	suna [100.0%]	sunf [100.0%]	sunc [100.0%]	suni [100.0%]		
9	sunh [100.0%]	sunb [100.0%]	sunc [100.0%]	sunf [100.0%]		
10	sunb [100.0%]	suni [100.0%]	sunc [100.0%]	suna [100.0%]	sunh [100.0%]	
11	suni [99.8%]	suna [100.0%]	sunf [100.0%]	sunc [100.0%]	sunh [100.0%]	
12	sunc [100.0%]	sunf [100.0%]	suna [100.0%]	sunh [100.0%]	sunb [100.0%]	
13	sunc [100.0%]	sunf [100.0%]	sunb [100.0%]	suna [100.0%]	sunh [100.0%]	suni [100.0%]
14	sunf [100.0%]	sunb [100.0%]	suni [100.2%]	suna [99.1%]	sunh [100.0%]	sunc [100.0%]
15	sunf [100.2%]	suna [78.9%]	sunb [100.0%]	sunc [100.0%]	suni [100.0%]	sunh [100.0%]

Table D.6: Timing of tt4 on client suna

Layout:tt5 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	44.25	12.96 [29%]	28.99 [66%]	2.06 [5%]	68.3	1
2	50.98	18.38 [36%]	28.86 [57%]	3.54 [7%]	49.9	2
2	48.76	16.46 [34%]	28.92 [59%]	3.16 [6%]	52.4	3
3	44.53	16.54 [37%]	22.83 [51%]	4.48 [10%]	51.6	4
3	42.84	16.00 [37%]	20.74 [48%]	4.72 [11%]	49.9	5
3	42.98	17.26 [40%]	20.82 [48%]	4.30 [10%]	49.6	6
4	38.42	15.48 [40%]	16.30 [42%]	4.02 [10%]	54.2	7
4	42.51	16.22 [38%]	19.69 [46%]	5.72 [13%]	52.1	8
4	39.00	16.54 [42%]	16.24 [42%]	5.62 [14%]	51.5	9
5	40.57	18.82 [46%]	14.01 [35%]	6.88 [17%]	48.9	10
5	39.56	16.40 [41%]	14.96 [38%]	7.66 [19%]	51.9	11
5	43.55	22.18 [51%]	13.26 [30%]	7.29 [17%]	46.6	12
6	43.26	19.28 [45%]	13.72 [32%]	9.56 [22%]	49.3	13
6	39.50	17.38 [44%]	12.56 [32%]	8.90 [23%]	49.9	14
6	50.82	23.52 [46%]	14.94 [29%]	11.40 [22%]	40.7	15

Note	Processors [cpu usage]					
1	sunf [100.0%]	sunh [97.8%]				
2	sunb [100.1%]	suna [100.0%]				
3	sunc [100.1%]	suni [100.0%]				
4	sunh [100.0%]	sunf [92.9%]	sunc [100.0%]			
5	suni [100.0%]	suna [100.0%]	sunb [100.0%]			
6	sunc [100.0%]	sunf [96.0%]	sunb [100.0%]			
7	suna [100.0%]	sunf [100.0%]	sunc [100.0%]	sunh [100.0%]		
8	sunb [99.5%]	suna [85.8%]	sunf [100.0%]	sunc [100.0%]		
9	suni [100.0%]	sunh [100.0%]	sunc [100.0%]	sunf [100.0%]		
10	sunf [100.2%]	suni [100.0%]	sunh [99.6%]	sunc [99.8%]	suna [100.0%]	
11	sunb [100.0%]	sunc [100.0%]	sunf [100.0%]	suni [93.4%]	sunh [100.0%]	
12	sunf [100.0%]	suna [100.0%]	sunh [100.0%]	suni [100.0%]	sunc [100.0%]	
13	sunc [88.7%]	sunb [100.0%]	suna [100.0%]	sunh [100.0%]	suni [97.5%]	sunf [100.0%]
14	sunb [100.0%]	sunf [100.0%]	suni [100.0%]	sunh [100.0%]	suna [100.0%]	sunc [100.0%]
15	sunh [89.2%]	sunc [100.0%]	suna [100.0%]	sunb [96.5%]	suni [100.2%]	sunf [100.0%]

Table D.7: Timing of tt5 on client suna

Layout:tt6 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	63.84	21.08 [33%]	38.70 [61%]	3.50 [5%]	53.3	1
2	61.82	21.62 [35%]	36.74 [59%]	3.20 [5%]	53.0	2
2	66.86	22.64 [34%]	39.52 [59%]	4.24 [6%]	49.2	3
3	57.82	24.82 [43%]	26.28 [45%]	6.00 [10%]	46.8	4
3	48.52	17.02 [35%]	26.32 [54%]	4.62 [10%]	64.7	5
3	54.64	21.60 [40%]	26.36 [48%]	6.28 [11%]	50.6	6
4	53.02	23.68 [45%]	20.92 [39%]	7.86 [15%]	49.5	7
4	50.72	21.29 [42%]	20.95 [41%]	7.86 [15%]	51.7	8
4	49.50	21.14 [43%]	20.84 [42%]	6.38 [13%]	56.3	9
5	54.22	23.48 [43%]	17.70 [33%]	12.40 [23%]	48.4	10
5	49.74	21.86 [44%]	17.86 [36%]	9.26 [19%]	51.8	11
5	54.54	22.96 [42%]	17.02 [31%]	13.14 [24%]	46.5	12
6	43.18	16.82 [39%]	15.42 [36%]	8.44 [20%]	62.3	13
6	51.30	23.18 [45%]	15.52 [30%]	11.82 [23%]	49.1	14
6	55.70	27.62 [50%]	15.60 [28%]	11.80 [21%]	46.6	15

Note	Processors [cpu usage]					
1	sunc [96.1%]	sunh [100.0%]				
2	suni [100.0%]	sunb [100.0%]				
3	sunc [100.0%]	suna [89.4%]				
4	suni [100.0%]	sunf [100.0%]	sunb [100.0%]			
5	sunc [100.0%]	sunh [100.0%]	suna [100.0%]			
6	suna [100.0%]	sunb [100.0%]	suni [100.0%]			
7	suna [100.0%]	sunb [100.0%]	suni [100.0%]	sunc [100.0%]		
8	sunh [100.0%]	sunf [100.0%]	suni [100.0%]	sunb [100.0%]		
9	sunc [100.0%]	suna [100.0%]	sunb [100.0%]	sunf [100.0%]		
10	sunh [100.0%]	sunb [100.0%]	suna [100.0%]	sunf [100.0%]	sunc [99.5%]	
11	suni [100.0%]	sunc [100.1%]	sunf [97.1%]	suna [100.0%]	sunb [100.0%]	
12	sunb [100.0%]	sunc [100.0%]	suna [100.0%]	sunf [100.0%]	sunh [100.0%]	
13	suna [100.0%]	sunf [99.6%]	sunh [100.0%]	suni [100.0%]	sunb [100.0%]	sunc [100.0%]
14	suni [100.0%]	sunc [100.0%]	sunf [100.0%]	sunb [100.0%]	sunh [100.0%]	suna [100.0%]
15	suni [100.0%]	sunc [100.0%]	sunf [100.0%]	sunb [100.0%]	suna [100.0%]	sunh [98.8%]

Table D.8: Timing of tt6 on client suna

Layout:tt10 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	103.33	26.00 [25%]	66.29 [64%]	5.82 [6%]	74.8	1
2	105.20	26.08 [25%]	68.06 [65%]	5.78 [5%]	74.1	2
2	109.98	28.50 [26%]	70.84 [64%]	5.46 [5%]	72.4	3
3	87.68	25.72 [29%]	46.32 [53%]	10.46 [12%]	73.0	4
3	85.36	26.04 [31%]	46.26 [54%]	7.88 [9%]	76.0	5
3	85.90	26.30 [31%]	46.32 [54%]	8.02 [9%]	75.7	6
4	79.18	28.08 [35%]	35.46 [45%]	10.28 [13%]	73.1	7
4	78.74	25.42 [32%]	37.56 [48%]	10.54 [13%]	76.0	8
4	77.65	25.85 [33%]	36.18 [47%]	10.32 [13%]	76.7	9
5	74.40	25.54 [34%]	30.46 [41%]	13.06 [18%]	77.5	10
5	74.61	25.72 [34%]	30.60 [41%]	13.09 [18%]	76.3	11
5	75.82	25.96 [34%]	31.56 [42%]	12.94 [17%]	76.2	12
6	79.68	25.44 [32%]	30.92 [39%]	17.88 [22%]	73.4	13
6	73.96	25.50 [34%]	28.02 [38%]	15.08 [20%]	78.0	14
6	80.16	26.12 [33%]	33.16 [41%]	15.26 [19%]	76.1	15

Note	Processors [cpu usage]					
1	sunf [100.0%]	sunc [100.0%]				
2	sune [98.2%]	sund [99.7%]				
3	sung [95.2%]	sunb [100.0%]				
4	sunc [100.0%]	sune [100.0%]	sunb [99.8%]			
5	sung [100.0%]	sune [97.0%]	sunf [100.0%]			
6	sunc [99.9%]	sund [100.0%]	sunb [97.1%]			
7	sung [100.0%]	sunc [100.0%]	sunb [100.0%]	sund [100.1%]		
8	sunf [100.0%]	sung [96.2%]	sund [100.0%]	sunb [95.1%]		
9	sune [100.0%]	sunc [96.4%]	sunf [96.0%]	sung [100.0%]		
10	sunc [99.9%]	sund [100.0%]	sunf [100.0%]	sunb [95.6%]	sung [94.2%]	
11	sung [99.1%]	sune [100.0%]	sunf [100.0%]	sund [99.9%]	sunb [100.0%]	
12	sund [100.0%]	sunc [95.6%]	sunb [100.0%]	sunf [97.5%]	sune [99.1%]	
13	sune [99.8%]	sunb [92.7%]	sunc [100.0%]	sung [94.4%]	sund [100.0%]	sunf [99.8%]
14	sung [97.2%]	sunf [100.0%]	sune [100.0%]	sunc [100.0%]	sunb [100.0%]	sund [100.0%]
15	sunc [85.0%]	sund [100.0%]	sung [100.0%]	sune [100.0%]	sunf [94.0%]	sunb [100.0%]

Table D.9: Timing of tt10 on client suna

Layout:tt20 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	327.95	81.18 [25%]	223.09 [68%]	18.50 [6%]	85.5	1
2	325.82	81.34 [25%]	221.56 [68%]	17.76 [5%]	86.2	2
2	327.75	84.80 [26%]	218.07 [67%]	19.70 [6%]	82.2	3
3	265.88	81.54 [31%]	153.26 [58%]	25.90 [10%]	85.6	4
3	270.75	87.48 [32%]	150.31 [56%]	27.74 [10%]	80.8	5
3	266.78	83.96 [31%]	151.80 [57%]	25.82 [10%]	84.3	6
4	235.04	81.06 [34%]	115.14 [49%]	33.52 [14%]	85.7	7
4	238.70	80.38 [34%]	116.92 [49%]	36.06 [15%]	84.0	8
4	243.72	83.94 [34%]	119.04 [49%]	35.42 [15%]	82.2	9
5	223.64	80.50 [36%]	96.58 [43%]	41.18 [18%]	85.4	10
5	223.43	80.20 [36%]	96.61 [43%]	41.28 [18%]	85.9	11
5	224.31	82.32 [37%]	95.31 [42%]	41.34 [18%]	84.6	12
6	234.02	84.75 [36%]	90.15 [39%]	53.58 [23%]	82.0	13
6	231.92	82.12 [35%]	88.70 [38%]	55.72 [24%]	82.2	14
6	225.93	84.38 [37%]	86.58 [38%]	49.66 [22%]	83.4	15

Note	Processors [cpu usage]					
1	sunf [97.9%]	sune [99.9%]				
2	sung [98.4%]	sunb [98.7%]				
3	sunc [99.3%]	sund [98.4%]				
4	sune [97.5%]	sung [97.0%]	sunf [99.5%]			
5	sunc [99.3%]	sund [99.9%]	sunb [100.0%]			
6	sunf [98.4%]	sung [99.7%]	sune [100.0%]			
7	sune [99.5%]	sunc [98.7%]	sunf [98.6%]	sund [98.4%]		
8	sunb [98.7%]	sung [98.2%]	sund [98.5%]	sune [99.9%]		
9	sunf [98.8%]	sunb [97.7%]	sund [98.8%]	sune [99.9%]		
10	sune [98.2%]	sung [98.1%]	sunb [98.0%]	sunf [99.8%]	sunc [99.8%]	
11	sund [98.1%]	sung [98.0%]	sunb [98.1%]	sunc [100.0%]	sunf [100.0%]	
12	sunf [99.9%]	sune [98.2%]	sunc [100.0%]	sung [98.5%]	sunb [97.2%]	
13	sung [97.8%]	sune [93.1%]	sund [99.3%]	sunb [99.5%]	sunc [99.4%]	sunf [99.1%]
14	sund [94.4%]	sunf [100.0%]	sunc [100.0%]	sune [97.0%]	sung [97.3%]	sunb [100.0%]
15	sunc [100.0%]	sung [96.4%]	sune [97.3%]	sunb [92.3%]	sunf [100.0%]	sund [99.1%]

Table D.10: Timing of tt20 on client suna

Layout:shift_30 Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	256.18	78.88 [31%]	158.30 [62%]	13.80 [5%]	86.2	1
2	258.88	81.82 [32%]	157.74 [61%]	13.84 [5%]	83.3	2
2	256.27	78.18 [31%]	159.25 [62%]	13.50 [5%]	87.1	3
3	213.40	79.10 [37%]	109.10 [51%]	19.98 [9%]	85.6	4
3	216.90	82.16 [38%]	109.68 [51%]	19.82 [9%]	83.7	5
3	213.53	78.18 [37%]	110.37 [52%]	19.74 [9%]	86.9	6
4	196.60	78.26 [40%]	86.86 [44%]	25.78 [13%]	85.9	7
4	200.24	82.50 [41%]	86.68 [43%]	25.54 [13%]	83.9	8
4	198.70	81.02 [41%]	85.84 [43%]	26.46 [13%]	83.2	9
5	194.58	78.32 [40%]	74.84 [38%]	35.92 [18%]	83.1	10
5	191.43	80.94 [42%]	73.07 [38%]	32.04 [17%]	84.4	11
5	187.32	78.94 [42%]	70.96 [38%]	32.00 [17%]	84.9	12
6	189.42	80.24 [42%]	63.54 [34%]	40.24 [21%]	83.2	13
6	188.45	81.02 [43%]	63.33 [34%]	38.52 [20%]	83.6	14
6	182.66	78.52 [43%]	60.90 [33%]	37.80 [21%]	85.5	15

Note	Processors [cpu usage]					
1	sung [98.2%]	sunb [97.2%]				
2	sune [98.7%]	sunf [97.4%]				
3	sung [98.1%]	sund [98.3%]				
4	sung [99.3%]	sunf [100.0%]	sune [98.5%]			
5	sunc [99.2%]	sunb [100.0%]	sune [98.4%]			
6	sung [98.8%]	sunf [100.0%]	sund [100.0%]			
7	sunb [98.5%]	sung [98.3%]	sunf [99.7%]	sunc [100.0%]		
8	sund [100.0%]	sung [98.4%]	sunb [100.0%]	sune [98.0%]		
9	sung [99.8%]	sunb [99.3%]	sune [100.0%]	sund [100.0%]		
10	sunc [98.2%]	sung [100.0%]	sunb [99.9%]	sund [97.8%]	sunf [100.0%]	
11	sunf [98.0%]	sung [100.0%]	sund [100.0%]	sunb [99.3%]	sune [100.0%]	
12	sune [100.0%]	sunb [100.0%]	sund [100.0%]	sung [100.0%]	sunc [100.0%]	
13	sund [97.8%]	sunb [99.9%]	sunf [100.0%]	sunc [99.1%]	sune [100.0%]	sung [100.0%]
14	sund [100.0%]	sunf [98.4%]	sunc [97.6%]	sune [100.0%]	sunb [100.0%]	sung [100.0%]
15	sung [100.0%]	sunb [99.6%]	sunf [100.0%]	sunc [100.0%]	sune [100.0%]	sund [99.2%]

Table D.11: Timing of shift_30 on client suna

Layout:bit Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	725.53	213.46 [29%]	464.59 [64%]	44.36 [6%]	87.3	1
2	736.72	220.04 [30%]	466.22 [63%]	47.76 [6%]	84.7	2
2	719.96	216.68 [30%]	452.88 [63%]	48.68 [7%]	85.4	3
3	592.68	221.10 [37%]	307.28 [52%]	63.20 [11%]	84.7	4
3	598.31	215.02 [36%]	317.62 [53%]	64.39 [11%]	86.6	5
3	595.44	219.06 [37%]	308.94 [52%]	66.36 [11%]	84.1	6
4	537.19	215.26 [40%]	231.06 [43%]	88.53 [16%]	83.4	7
4	541.01	221.42 [41%]	236.50 [44%]	79.91 [15%]	83.9	8
4	544.55	220.14 [40%]	240.51 [44%]	82.38 [15%]	84.1	9
5	524.88	224.78 [43%]	194.32 [37%]	104.58 [20%]	81.8	10
5	519.31	224.72 [43%]	199.69 [38%]	93.96 [18%]	83.9	11
5	533.66	223.56 [42%]	207.90 [39%]	100.28 [19%]	82.6	12
6	514.12	218.06 [42%]	182.20 [35%]	112.74 [22%]	84.9	13
6	506.85	219.96 [43%]	173.01 [34%]	112.54 [22%]	84.2	14
6	520.25	224.74 [43%]	170.82 [33%]	122.71 [24%]	81.2	15

Note	Processors [cpu usage]					
1	sune [90.2%]	sunc [90.6%]				
2	sunf [90.4%]	sund [92.2%]				
3	sunc [92.3%]	sunb [92.1%]				
4	sund [94.6%]	sung [93.9%]	sune [94.8%]			
5	sunc [90.6%]	sunb [94.1%]	sunf [93.2%]			
6	sund [93.8%]	sune [93.4%]	sung [92.2%]			
7	sung [95.3%]	sund [96.0%]	sunc [96.5%]	sunf [97.4%]		
8	sune [98.2%]	sunb [97.6%]	sunf [94.0%]	sund [94.4%]		
9	sung [95.6%]	sunc [97.3%]	sunf [92.2%]	sunb [96.3%]		
10	sunf [97.8%]	sunb [98.1%]	sund [95.4%]	sune [95.9%]	sunc [94.9%]	
11	sunc [91.4%]	sunb [95.0%]	sune [95.7%]	sund [95.7%]	sung [99.0%]	
12	sund [89.8%]	sunb [95.3%]	sunf [92.6%]	sune [93.9%]	sung [97.6%]	
13	sund [98.0%]	sunf [90.7%]	sunb [96.0%]	sunc [98.2%]	sune [88.3%]	sung [97.7%]
14	sunb [96.5%]	sunc [95.3%]	sund [96.8%]	sunf [91.5%]	sune [98.3%]	sung [99.6%]
15	sund [98.2%]	sunc [93.5%]	sunf [95.3%]	sunb [97.9%]	sung [98.1%]	sune [98.0%]

Table D.12: Timing of bit on client suna

Layout:mem Client: suna						
Number of Processors	Total (sec)	Serial (sec)	Parallel (sec)	Comm (sec)	Cpu Usage	Note
2	1031.73	295.38 [29%]	654.30 [63%]	72.87 [7%]	78.6	1
3	1175.25	355.74 [30%]	663.24 [56%]	147.95 [13%]	61.9	2
3	814.02	287.24 [35%]	419.50 [52%]	99.52 [12%]	79.3	3
3	808.04	286.05 [35%]	396.82 [49%]	113.83 [14%]	75.7	4
4	757.86	282.16 [37%]	334.10 [44%]	132.51 [17%]	78.3	5
4	774.28	282.30 [36%]	336.26 [43%]	147.40 [19%]	75.9	6
4	761.73	288.60 [38%]	316.96 [42%]	147.75 [19%]	74.5	7
5	747.53	276.58 [37%]	290.26 [39%]	170.95 [23%]	74.3	8
5	773.92	273.98 [35%]	315.52 [41%]	175.56 [23%]	74.7	9
5	858.15	298.58 [35%]	375.94 [44%]	174.73 [20%]	72.9	10
6	813.09	298.43 [37%]	284.68 [35%]	220.22 [27%]	69.0	11
6	819.62	312.58 [38%]	273.36 [33%]	222.76 [27%]	68.7	12
6	842.55	332.15 [39%]	247.34 [29%]	252.79 [30%]	63.9	13

Note	Processors [cpu usage]					
1	sung [77.6%]	sunc [78.5%]				
2	sunf [70.9%]	sunb [63.3%]	sunh [51.9%]			
3	sung [82.1%]	sunc [83.2%]	sunh [82.4%]			
4	sunf [87.4%]	sunh [87.0%]	sunb [84.9%]			
5	sunb [83.0%]	sund [81.7%]	sung [83.4%]	sunf [79.7%]		
6	sunh [81.1%]	sunf [83.7%]	sund [80.9%]	sunb [78.0%]		
7	sunc [82.7%]	sung [87.4%]	sunh [88.9%]	sunb [85.6%]		
8	sunf [81.2%]	sunh [81.0%]	sunb [81.9%]	sund [83.4%]	sunc [65.7%]	
9	sung [88.4%]	sunb [69.3%]	sunh [86.0%]	sund [85.2%]	sunf [76.1%]	
10	sunc [60.3%]	sung [72.2%]	sund [73.9%]	sunh [73.0%]	sunb [67.7%]	
11	sund [77.4%]	sunb [81.2%]	sunf [80.2%]	sunh [77.5%]	sung [83.7%]	sunc [60.2%]
12	sunh [71.8%]	sung [87.4%]	sunb [84.7%]	sund [85.6%]	sunf [84.9%]	sunc [81.7%]
13	sund [82.2%]	sunf [82.1%]	sunb [81.5%]	sung [84.0%]	sunh [76.4%]	sunc [77.8%]

Table D.13: Timing of mem on client suna

CURRICULUM VITAE

RODRIGUE G. BYRNE

PERSONAL DATA

45 Prince Wales St.
St. John's, NFLD
A1C 1N2

BORN: August 7, 1960
Single
Canadian

EDUCATION

1984	Bachelors Degree in Computer Science (Honours) (Memorial) Bachelors Degree in Electrical Engineering. (Memorial)

AWARDS

1986	B.C. Scholarship
1983	Digital Equipment of Canada Award of Merit
1982	Hudson Bay Company Award in Computer Science
1977	Grade XI - Highest Marks Chemistry and Physics Award Grade XI - Honorable Mention - Physics Competition Award Grade XI - Judge Higgins Memorial Scholarship Grade X - Confederation Scholarship

EMPLOYMENT

Sept 1987 - Present	Computer Science Department Memorial University of Newfoundland Position: Lecturer
Sept 1986 - August 1987	Faculty of Engineering and Applied Science Memorial University of Newfoundland Position: Lecturer
Sept 1984-Aug 1985	Computer Science Department Memorial University of Newfoundland Position: Research Assistant Worked on VLSI CAD systems.

PUBLICATIONS

Byrne, R.,*Fred: An Implementation of an Arbitary 2-Place Quarternay Function in CMOS* in 1986 Canadian Conference on VLSI, Montreal. p145-150

PARTIAL COPYRIGHT LICENCE

A DISTRIBUTED LAYOUT COMPACTOR

Author: [redacted]

RODRIGUE BYRNE

May 24, 1988