PARALLEL ALGORITHMS FOR LAYOUT VERIFICATION

BY

KY MACPHERSON

B.S., University of Alabama, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for his direction and assistance with my research; John Chandy and the rest of my office-mates for sharing their expertise with the ProperCAD programming environment; and my family for their support and encouragement.

This research was supported by the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office.

DEDICATION

In memory of Alisa

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

As VLSI technology advances allow smaller feature sizes so that more devices can be fit into a given chip area, and as the complexity of designs continues to grow, distributing the workload and memory requirements among different processors for computationally intensive tasks such as VLSI layout design rule checking (DRC) becomes increasingly important. In this thesis, we intend to show that a combination of data parallelism and task parallelism can be used to implement VLSI geometry design rule checking on a variety of multiprocessor platforms.

1.1 Overview of ProperCAD

Parallel processing for CAD has become an important research area, as the increasing complexity of VLSI circuits continues to place higher demands on VLSI CAD tools. A major limitation with almost all previous work in parallel CAD is that the parallel algorithms have been targeted to run on specific machines or architectures. The lack of portability of the algorithms is a severe limitation on their usefulness to the VLSI CAD community as a whole.

Another important problem with the implementation of parallel CAD tools is the substantial time investment required for development. The result is that parallel CAD software is more costly to develop than sequential algorithms. Furthermore, given the fast pace of progress in the development and improvement of sequential algorithms for CAD applications, by the time a parallel implementation of a given algorithm is available, significant improvements in serial algorithms may offer better quality results and/or comparable performance.

Several recent attempts have been made to develop machine-independent parallel programming environments [1]. Some of these implementations introduced entirely new programming languages to facilitate portability across several types of architectures. Some more recent efforts have used a library-based approach to allow integration with popular programming languages and compatibility with existing compilers and other development tools.

The ProperCAD project at the University of Illinois by Banerjee et al. addresses both the need for portability across parallel machines and the capability for easy integration of existing sequential algorithms in the design of parallel algorithms for CAD [2]-[5]. ProperCAD differs from earlier approaches to parallel programming efforts in the following ways: It targets existing architectures that support medium-grain parallelism; the constructs used to express parallelism are native constructs provided by the C++ language;

no extensions to the language are used, so that existing compliers and debuggers can be used for development; and portability across a wide variety of parallel architectures is stressed.

ProperCAD utilizes two distinct interfaces to simultaneously maximize portability and efficiency, as shown in Figure 1.1. The upper level of the library, the actor interface, provides a high-level interface to assist application development by limiting complexity through abstraction and encapsulation, without limiting parallelism. The abstract parallel architecture level of the library provides a set of interfaces and implementations that can be used to describe and utilize resources needed by any parallel application across a wide variety of architectures, including processor threads, synchronization primitives, and memory management techniques.

The ProperCAD actor interface is based closely on the Actor paradigm and supports such features as aggregates and meta-programmability. The fundamental object in the Actor paradigm is the actor, which is an object that communicates with other actors by sending messages. All actions that an actor performs are in response to other messages; thus, the model is message driven.

An important facet of the Actor model is the lack of explicit sequencing primitives. Synchronization is implicit and derives from the message reception serialization property of the model: while an actor is processing a message, additional messages are not received. Because there are no explicit synchronization primitives, an actor cannot block or suspend itself in anticipation of a future event.

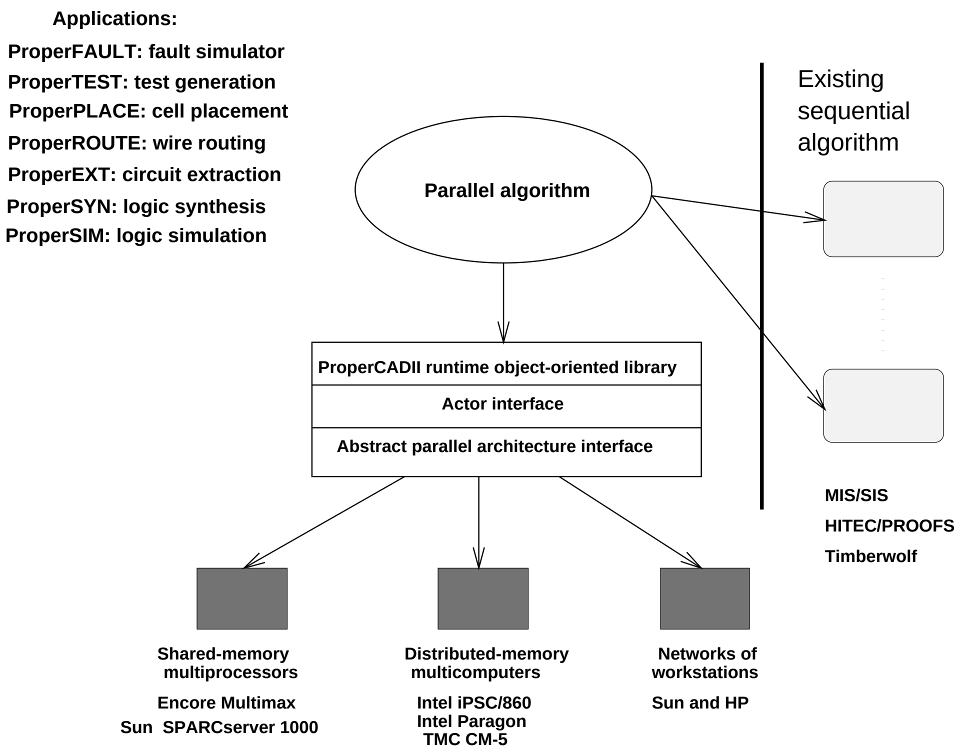


Figure 1.1: Overview of the ProperCAD II project

In the Actor paradigm, *continuations* fill the role of parallel function calls. Continuation calls differ from traditional C++ member function calls in the following ways: Continuation execution occurs asynchronously, so that the actor invoking the continuation can overlap computations with the time required for the communication to take place, and continuation calls do not return a value, due to their asynchronous nature. Figure 1.2 shows a continuation call from one thread to another, in which a return value may eventually be returned from the second thread to the first by means of another continuation.

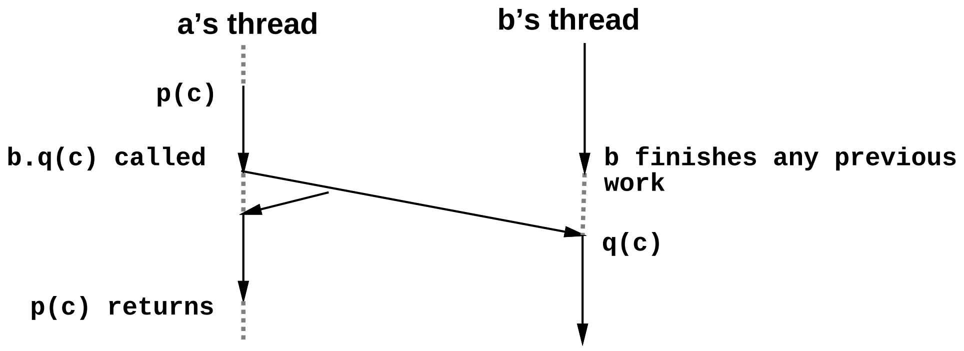


Figure 1.2: Continuation passing

1.2 Introduction to Layout Verification

To guarantee that a circuit can be reliably fabricated, it is necessary to impose a set of *design rules* on the layout geometry. The basic physics of semiconductor circuits give rise to some of the constraints, while others may be necessary as a result of imprecision in the fabrication processes. Therefore, for a given manufacturing process, a set of design rules will be defined to guarantee an acceptable yield on circuits fabricated with the technology. Some examples of commonly used design rules are presented in Figure 1.3.

In ProperDRC, the DRC tool developed with the ProperCAD programming model, the circuit geometry is dynamically partitioned among processor clusters. The steps required to test the circuit geometry against the individual design rules are broken down into a set of elementary tasks, which are partitioned among the processors in the cluster. There are, therefore, two distinct levels of parallelism: data parallelism at the cluster level, and task parallelism at the processor level.

Among VLSI CAD applications, the DRC problem is especially well suited for parallelization in that there is no need for communication among processors while the geometry

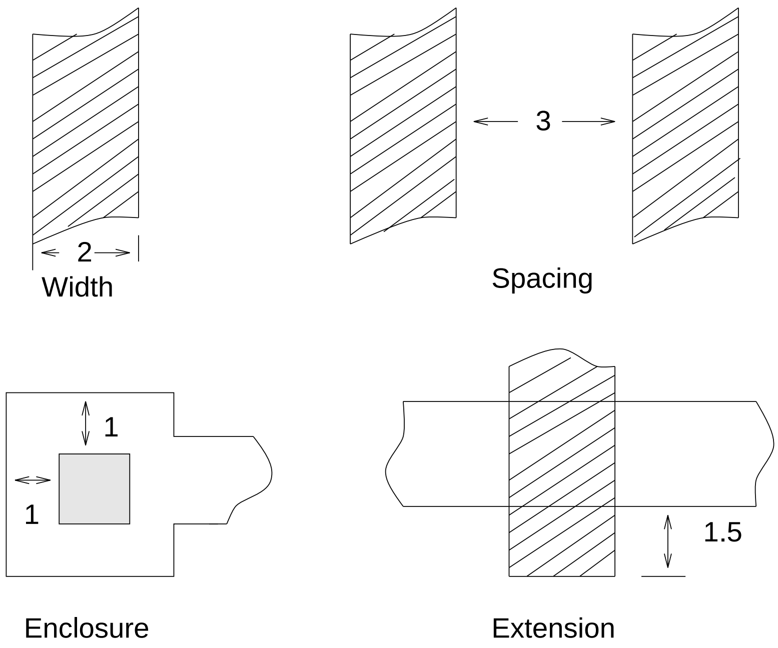


Figure 1.3: Typical design rules in VLSI layouts

checks are being performed. Intracluster communication may be necessary upon the completion of individual tasks, so that subsequent tasks may be spawned on other processors. Intercluster communication occurs only during the partitioning phase of the DRC, when the circuit is being partitioned among processors, and during the merge phase, when the individual cluster results are combined to produce the DRC results for the entire circuit. Both of these communication phases are one-time costs, and will take a negligible amount of time compared to the geometry rule checking, for an adequately large problem size. Ideally, linear speedups are attainable, due to the aforementioned lack of communication, if the load is properly balanced both among clusters and among processors inside each cluster.

1.3 Thesis Outline

This thesis is organized as follows: Chapter 2 will describe the details of the serial algorithm for design rule checking. Chapter 3 will discuss related work in parallel design rule checking algorithms. Details of the parallel DRC algorithm and of the ProperDRC implementation are described in Chapter 4. Performance results for ProperDRC are presented in Chapter 5. Chapter 6 contains an analysis of the performance of the parallel DRC, as well as an analysis of a novel serial DRC algorithm presented in this thesis. Chapter 7 summarizes the work in parallel DRC performed in this research. The appendices contain specific details and examples of the use of ProperDRC.

2. SERIAL DESIGN RULE CHECKING

2.1 Geometric Representation Schemes

The manner in which the design rule checks are performed may be partly dependent on the representation scheme used to describe the circuit geometry. No single scheme is best suited to all of the different kinds of operations that are involved in layout analysis. The amount of data required to describe the layout of a chip of moderate or large size is so great that designers of layout analysis programs must usually choose a single representation and tolerate more complicated algorithms or suboptimal performance in some parts of their systems [6]. Some commonly used forms of layout representation are polygon, pixelmap, corner-based, and edge representations.

The polygon representation uses the vertices of a geometry feature to describe it. There is a one-to-one mapping between physical features in the circuit area and polygon structures in the circuit representation. The polygon form is useful for the storage of mask information, but is inconvenient for computation.

Only layout regions whose corners have integer coordinates and whose edges are all horizontal or vertical may be represented in the pixelmap form. A pixel represents a single unit area in the circuit area. Each pixel contains several bits, each of which corresponds to a mask layer. Algorithms that operate on a pixelmap representation are usually simple to understand and implement; however, the large memory requirement for this representation often makes them too expensive to be practical.

The corner- or tile-based representation also applies only to Manhattan layouts, in which the edges of any geometry feature are parallel to either the x- or the y-axis. The basis for the representation, the *corner*, is defined to be any point at which two or more edges intersect. These corners divide the plane into rectangular *tiles*. Each tile is labeled to indicate which layers are present. The tile-based representation is less expensive than the pixelmap representation, because single tiles can denote an area of any size. It can require more storage than the polygon representation, because intersecting geometry features in different layers contain additional corners that are not present in the original layers themselves.

The edge representation, like the polygon representation, uses lists of edges to represent geometry features. However, unlike the polygon information, edges corresponding to a single feature are not stored together, but are rearranged in such a way to facilitate computations on the mask layers. The masks of a Manhattan geometry can be represented by horizontal edges only, because the vertical edges can be reconstructed by

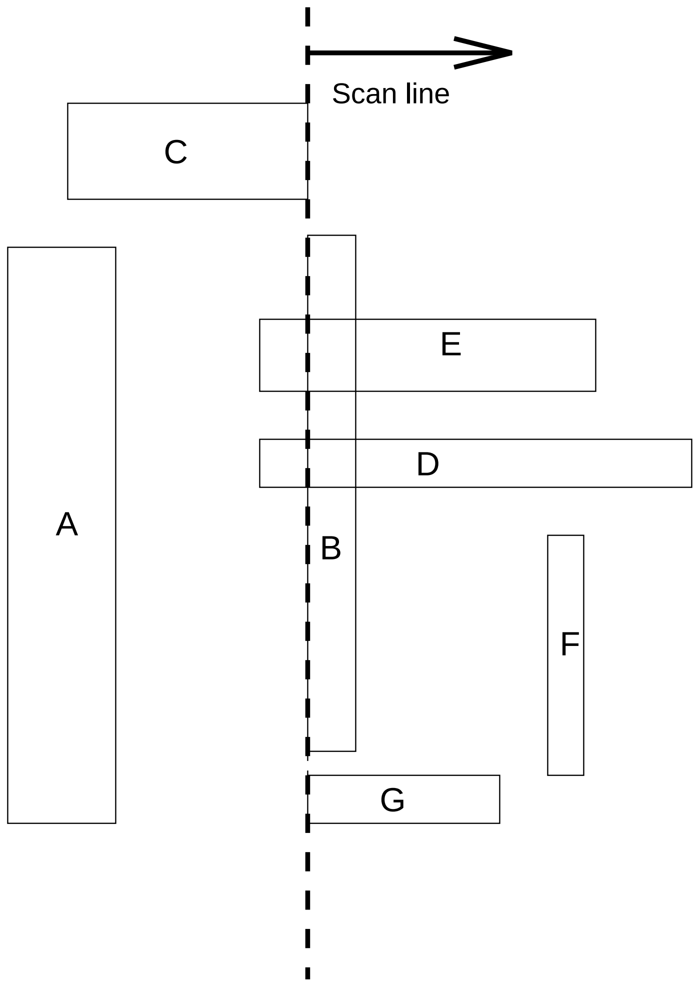


Figure 2.1: A scanline moving left to right

examining the opaqueness or transparency of the areas above and below each horizontal edge.

A data structure called the *scanline* is useful for performing operations on a geometry that uses an edge representation. The basic idea of a scanline algorithm is to sweep a vertical line across the edges that constitute a mask layer, as shown in Figure 2.1. Each horizontal location the scanline encounters is called a *scanline stop*. Only the edges that encounter the scanline are considered at a given time.

Scanline algorithms can be implemented in a space-efficient manner. An edge in the circuit area is brought in from secondary storage when its left endpoint touches the scanline and is removed from main memory when its right endpoint touches the scanline. For a circuit of size N, the amount of space necessary to implement a scanline algorithm is $O(\sqrt{N})$ [7]. Furthermore, the scanline stops can be restricted to locations on the circuit area that correspond to the left or right endpoint of an edge.

2.2 Useful Data Structures

There are several advantages to using the edge representation of VLSI layout geometry, as discussed in the previous section. It provides a good balance between a reasonable storage requirement and convenience of implementation of the algorithms necessary to perform design rule checking. The DRC algorithms presented here will utilize data structures appropriate for the edge representation form.

The *Edge* data structure is defined to represent a Manhattan geometry edge. The Edge structure has the following components: location, begin point, end point, and orientation. For a horizontal edge, the *location* would be the ordinate of the edge endpoints, the *begin* and *end points* would the the abscissas of the endpoints, and the *orientation* would be a value that differentiates between top edges and bottom edges of geometries. By using generic terms such as "location," "begin," and "end" to denote the edge endpoint coordinates, the same data structure can be used for both horizontal and vertical edges without loss of generality.

A data structure called an *EdgeSet* is defined for the purpose of maintaining sets of edges corresponding to each layer. The EdgeSet is implemented as a linked list. The following operations are defined on the EdgeSet data structure: Add, Sort, and Grow. The *Add* operation appends a new Edge to the EdgeSet. The *Sort* operation uses the Quicksort algorithm to order the Edges first by location, then by begin point, and finally by orientation. This Sort takes $O(N \log N)$ time for N edges. The *Grow* operation will be discussed later.

2.3 Task Graph Generation

Advances continue to occur in VLSI manufacturing technology; therefore, a practical DRC tool must have the flexibility to accommodate changes in design rules. ProperDRC reads a set of design rules from an input file and then generates a graph of the tasks required to perform the design rule checks. If it becomes necessary to change any of the design rules, only the input file must be modified; no changes to the source are necessary. The format used for the technology file, along with a description of the implementation of the MOSIS rules, is presented in Appendix A.

ProperDRC has the capability to test for violations of any of the following types of rules: Width, Spacing, Enclosure, Overlap, No-Overlap, and Extension. The *Width* rule is used to specify a minimum feature width for a given layer. The *Spacing* rule defines the minimum distance between geometries in two different layers, or between two geometries in a single layer. The *Enclosure* rule is used when one feature must surround another feature by a minimum distance on all sides. The *Overlap* rule is used when features in a given layer must always be overlapped by another layer. The *No-Overlap* rule serves just the opposite purpose, and is used to prevent two features in separate layers from occupying the same space. If one geometry feature must extend past the boundary of another feature by a certain minimum distance, the *Extension* rule is used.

Each of these rule checks is broken into one or more elementary tasks. The elementary tasks used by ProperDRC are Boolean operations between two layers, the Square-Test operation, the Grow operation, and Width/Spacing testing.

The majority of the rules equate to a single elementary task, as shown in Figure 2.2. The circles in the task graph represent the input layers, and the squares represent layers generated by the operation listed, which will contain all of the geometry edges that fail to pass the corresponding design rule.

The Enclosure and Extension rules require a series of three and ten elementary tasks, respectively, to test for rule violations. The task graphs corresponding to these two types of design rules are shown in Figure 2.3.

2.4 Elementary DRC Tasks

Boolean operations between layers are performed using Lauther's scanline algorithm [7]. The arguments to the function are the two input layers, and the result is an EdgeSet for the newly formed layer. This operation takes $O(N\ log\ N)$ time for N edges. The edges of the newly formed layer must be sorted, so that they may be used as arguments to

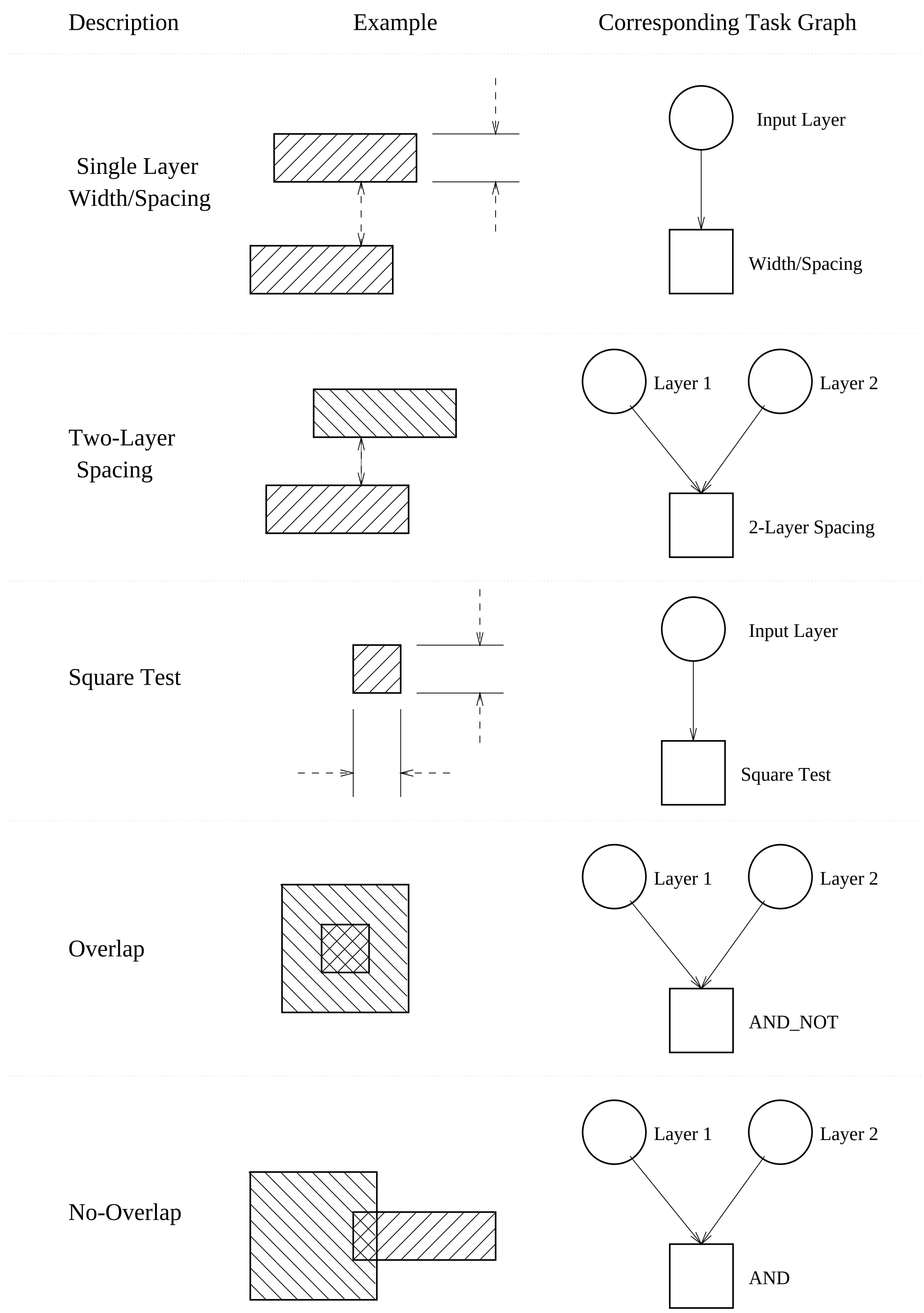


Figure 2.2: Design rules that correspond to one elementary task

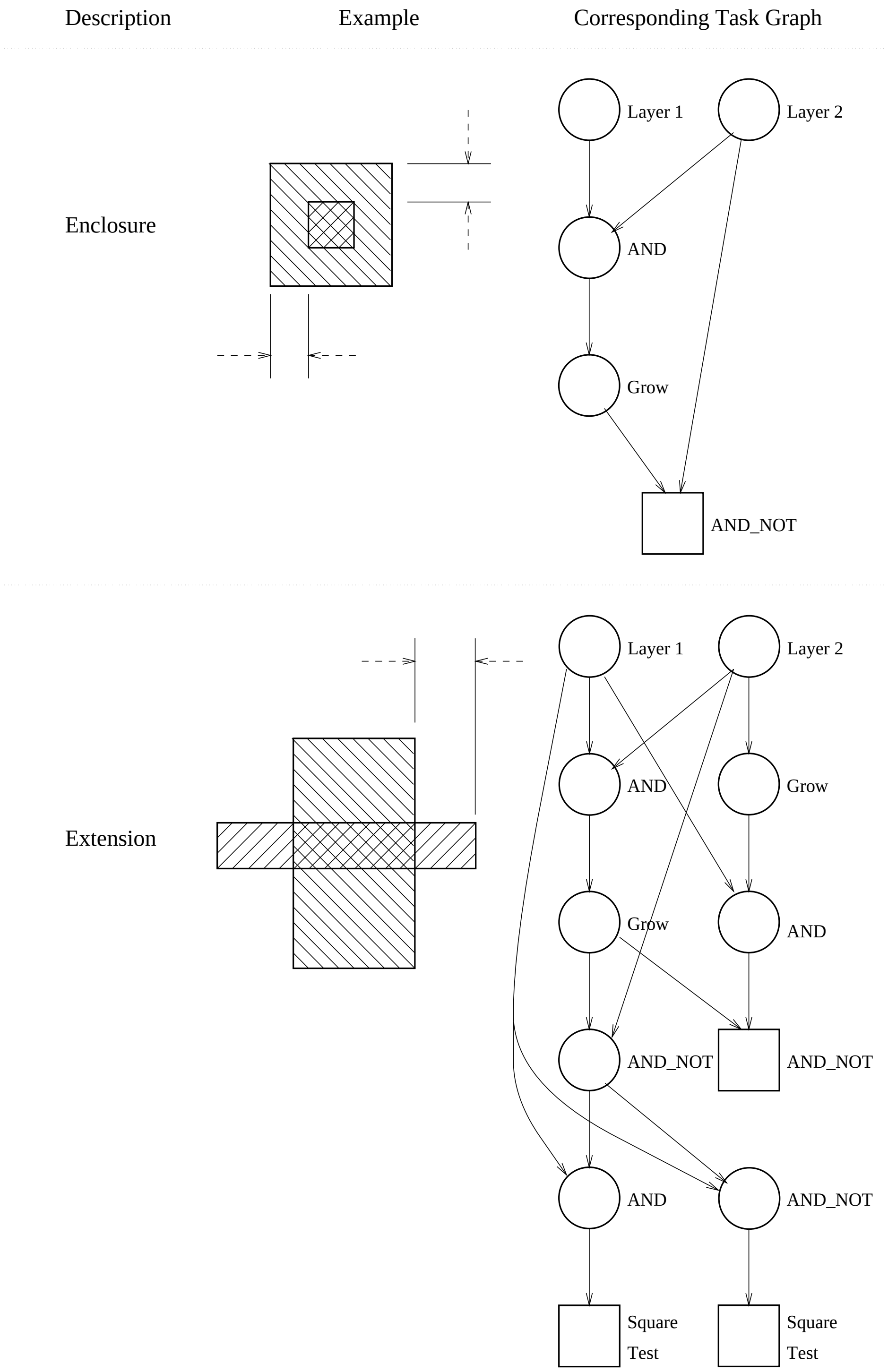


Figure 2.3: Tasks used to implement the Enclosure and Extension rules

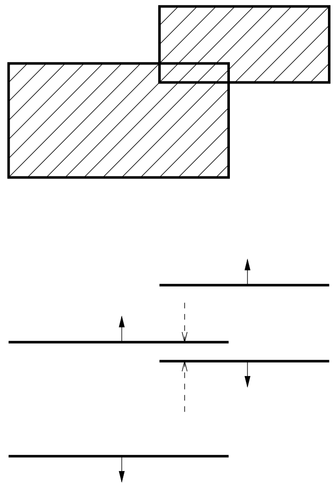


Figure 2.4: An apparent width violation from an unpainted set of edges

subsequent tasks. Szymanski and Van Wyk have demonstrated that the natural ordering for the output of a scanline operation can be exploited to perform the sort in $O(log\,N)$ time [8].

A special case of the Boolean operation is the paint operation, in which a scanline is passed over a single layer, and no Boolean operation is performed, *per se.* The purpose of the paint operation is to form a set of maximal, nonoverlapping edges. This procedure is necessary before passing a set of edges to the Width/Spacing operation, as Figure 2.4 demonstrates. The arrows placed on the edges denote the Orientation of the edge. In this case, performing the Width/Spacing check on the edges obtained directly from the original rectangles results in an apparent width violation, shown by the dashed lines.

Figure 2.5 shows the set of maximal, nonoverlapping edges produced by the paint operation. Note that the edges that produced the erroneous width violation in Figure 2.4 have been eliminated by the paint operation.

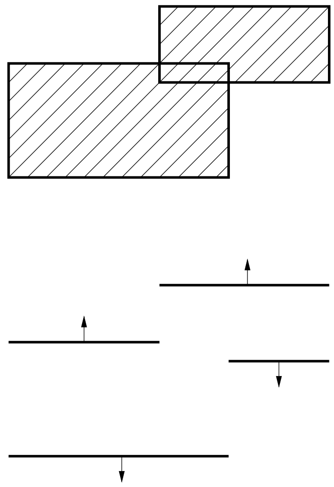


Figure 2.5: A painted set of edges

The Square-Test operation groups every edge of a given layer into pairs that form squares of a given size. This test is used to verify the size of features in the contact and via layers. This task is also used to implement the Extension test. This operation takes $O(N)$ time for N edges.

2.5 Grow Operation

The Grow operation is performed on a given layer and produces a new EdgeSet, in which every rectangle is expanded by a specified size. A modification of Lauther's Boolean mask scanline algorithm [7] is used to perform the grow operation. The implementation of the grow itself actually occurs at the output of the scanline operation, so a Boolean mask operation and Grow operation can be performed simultaneously, if necessary.

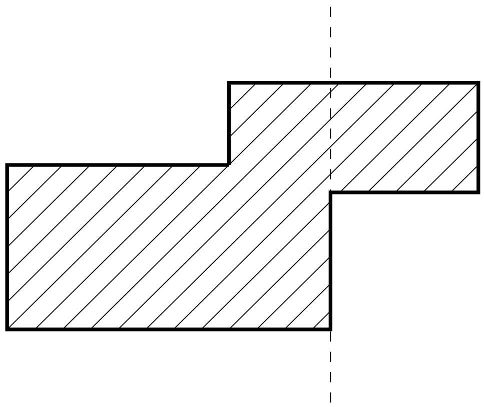
Two counters are used to maintain the presence or absence of a feature in the output layer at every stop along the output edge scanline. The difference between the counters can be used to indicate vertical edges on the output layer. Figure 2.6 shows how the counter values on the left- and right-hand sides of the scanline can be used to classify every edge endpoint as an interior or exterior polygon vertex.

Every new horizontal edge occurring on the output layer must be accompanied by a single vertical edge, either above or below the horizontal edge on the scanline. The C++ code used in ProperDRC to generate the vertical edges on the output scanline is shown in Figure 2.7.

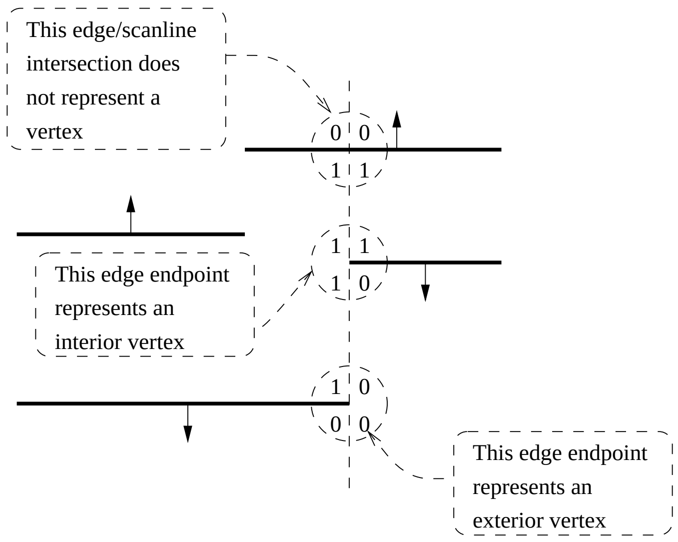
Comparing the counters to determine the location of the vertical edge allows the determination of whether the left endpoint of the horizontal edge represents an interior or exterior vertex of the geometry feature. Let G denote the distance by which the output layer is to be grown. For exterior vertices, the Edge begin value is decremented by G, whereas for interior vertices, the begin value is incremented by G. The location value of the Edge is also incremented or decremented by G, depending on the edge orientation.

The right endpoint of the output edge must likewise be accompanied by a vertical edge. A similar test is performed to determine whether the endpoint represents an interior or exterior vertex. The Edge end value is decremented or incremented by G based on the outcome of the test.

Figure 2.8 shows the code used to output the edges modified by the Grow operation. The result of a sample Grow operation is depicted in Figure 2.9. The vertical dashed lines represent the four scanline stops required to produce the result. The horizontal dashed lines are the original geometry edges, and the solid lines show the new edges. This Grow



(a) Original geometry intersected by scanline



(b) Classification of edge endpoints based on counter values

Figure 2.6: Differentiating between interior and exterior vertices

```
// initialize counters
lhsCounter[0]=0; rhsCounter[0]=0;
lhsCounter[1]=0; rhsCounter[1]=0;
newLValue=0;     newRValue=0;

while (not_exhausted(inputScanline))
{
    oldLValue=newLValue; oldRValue=newRValue;

    stop = nextInputEdge(inputScanline);

    for (lay=0;lay<=1;lay++)
        while( (nextInputEdge[lay] != NULL) &&
               (nextInputEdge[lay]->loc == stop->loc) )
        {
            // the Orientation value for an edge is
            // either -1 or +1, depending on whether the
            // edge is a top edge or bottom edge
            if(nextInputEdge[lay]->begin < x)
                lhsCounter[lay] -= nextInputEdge[lay]->orient;
            if(nextInputEdge[lay]->end > x)
                rhsCounter[lay] -= nextInputEdge[lay]->orient;
            nextInputEdge[lay]=inputScanline[lay].next();
        }

    // 'operation' performs the Boolean mask operation
    // on the counter values.  The return value is either 1 or 0
    newLValue=operation(lhsCounter);
    newRValue=operation(rhsCounter);

    // nonzero Ldifference means there is a horizontal output
    // edge on the left side of the current scanline stop
    Ldifference=oldLValue-newLValue;
    // Rdifference means the same thing for the right side of the scanline
    Rdifference=oldRValue-newRValue;

    // 'exterior_right_edge' indicates whether this scanline stop
    // represents an interior or exterior vertex for any output
    // edge that _Begins_ at this location
    if(Rdifference == (-1))
        exterior_right_edge = 1 - 2*oldLValue;
    else if(Rdifference == 1)
        exterior_right_edge = 1 - 2*newLValue;
    else
        exterior_right_edge = 0;

    // 'exterior_left_edge' indicates whether this scanline stop
    // represents an interior or exterior vertex for any output
    // edge that _Ends_ at this location
    if(Ldifference == (-1))
        exterior_left_edge = 1 - 2*oldRValue;
    else if(Ldifference == 1)
        exterior_left_edge = 1 - 2*newRValue;
    else
        exterior_left_edge = 0;

    OutputModifiedEdges();
}
```

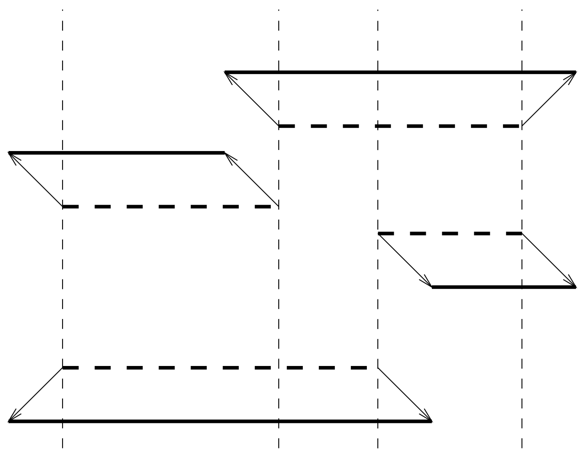
Figure 2.7: Determining position of vertical edges on the output scanline

```
OutputModifiedEdges()
{

    outputEdge = outputScanline.locate(stop->loc + grow*Rdifference);

    if ( (Rdifference != 0) &&
         (Ldifference != Rdifference) )
    {
        // add edge
        newEdge = (Edge *) _Alloc_Edge();
        newEdge->begin = x - grow*exterior_right_edge;
        newEdge->loc = stop->loc + grow*Rdifference;
        newEdge->orient = (Edge::edge_orientation) Rdifference;
        outputScanline.insert(newEdge);
    }

    outputEdge = outputScanline.locate(stop->loc + grow*Ldifference);

    while ( (outputEdge != NULL) &&
            (outputEdge->loc == stop->loc + grow*Ldifference) &&
            ( (int)outputEdge->orient == Ldifference) &&
            ( (Ldifference != Rdifference) ||
              (Rdifference == 0) ) )
    {
        // set edge endpoint
        outputEdge->end = x + grow*exterior_left_edge;
        // process edge
        ProcessEdge(outputEdge);
        outputScanline.removeEdge();
        outputEdge=outputScanline.peek();
    }
}
```

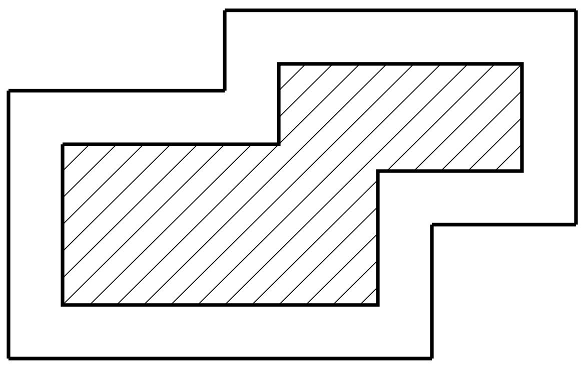
Figure 2.8: Modified scanline edge output routine to implement Grow operation

operation is an *orthogonal* grow, as defined by Preas and Lorenzetti [6], because the vertices of the resulting shape are on the angle bisectors of the vertices of the original geometries.

Because the implementation of the Grow operation as presented in this section can introduce overlapping edges in the resulting layer, a subsequent Width or Spacing check would have to be preceded by a paint operation. However, for all of the task graphs corresponding to the design rules described in the previous section, it should be noted that a Width or Spacing check never follows a Grow operation. Therefore, the paint operation may be forgone, to improve the efficiency of the DRC.



(a) Edges generated by the Grow operation



(b) Resulting geometry after Grow operation

Figure 2.9: Sample result of a Grow operation

2.6 Width/Spacing Test

There are two versions of the Width/Spacing test. One takes a single layer as an argument and tests all edges in that layer against other edges in the same layer for minimum width and spacing requirements. The second version takes two layers as arguments and tests all edges in the first layer against edges in the second layer, and vice versa, for spacing violations. The differences between these two versions are minor, so a single routine is used to perform both functions.

Several optimizations can be used to streamline the clearance checking algorithm. Only the endpoints of an edge must be tested. Ideally, this endpoint only needs to be tested against edges that lie within a circle whose center lies on the endpoint and whose radius is the minimum allowable clearance. For Manhattan edges, the search range can be further reduced by dividing the circle into quadrants. Only edges that lie in one quadrant of the circle require testing for spacing violations. This is illustrated in Figure 2.10. If a Width test is also being performed, a second quadrant of the circle must be tested for width violations. Figure 2.11 shows the regions that must be searched for width and spacing violations. Note that the arcs corresponding to the width and spacing quadrants may have different radii, if the minimum spacing and width distances are different.

The ProperDRC Width/Spacing test uses scanlines similar to the ones used in the Lauther algorithm. However, several scanlines must be kept in memory at a time, to test for violations between neighboring edges. Therefore, a new data structure, the Window, is introduced. A Window consists of a set of scanlines. In addition, the Window contains

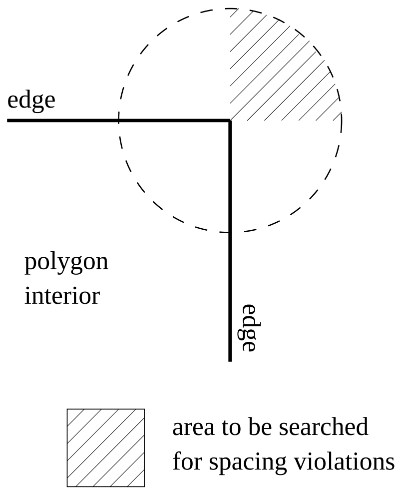


Figure 2.10: Ideal spacing search quadrant

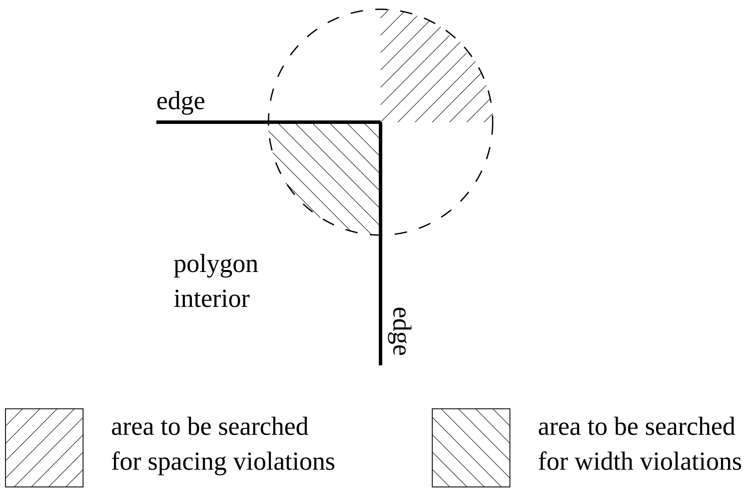


Figure 2.11: Ideal width/spacing search quadrants

additional storage for edges that lie parallel to the scanline, which must be generated by the Width/Spacing test routine.

The Window is essentially a swath cut through the circuit with a width of twice the maximum design rule interaction distance, ($DRID_{MAX}$), which is defined to be the greater of the minimum spacing distance and the minimum width distance for a given layer. This is used to ensure that a given edge is tested only for width/spacing violations against other edges that lie in close proximity to that edge. Figure 2.12 shows a Window of scanlines on a sample circuit. Note that the vertical edges are generated on the scanlines in the Window.

To make the Width/Spacing test even more efficient, it is desirable to compare edges within the Window that are in close proximity to each other. Therefore, the width/spacing routine essentially passes a second Window, perpendicular to the first one, across the length of the first Window. Figure 2.13 shows the intersecting horizontal and vertical Windows. The bold edges are the ones that lie inside the intersection of the two windows.

The result is that a given edge is tested only against edges that lie in a square whose size is twice the maximum design rule interaction distance on a side. The width/spacing testing can be further optimized by dividing the square into quadrants and restricting the searches to the appropriate quadrants. Figure 2.14 shows the regions surrounding a given edge endpoint that will be searched for edges that violate the width/spacing rules.

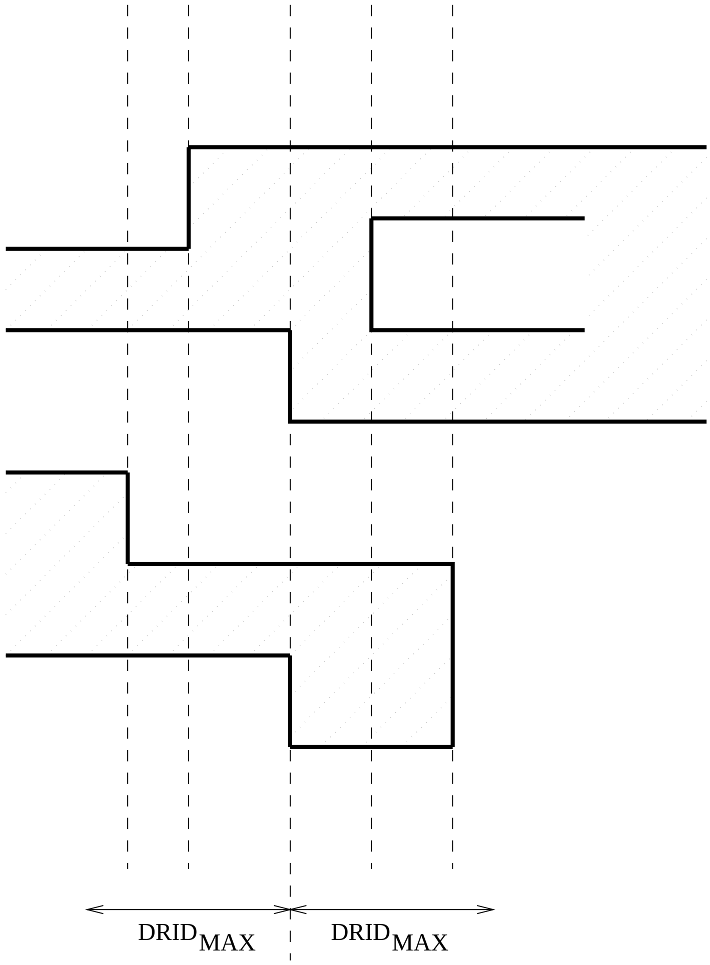


Figure 2.12: A vertical Window

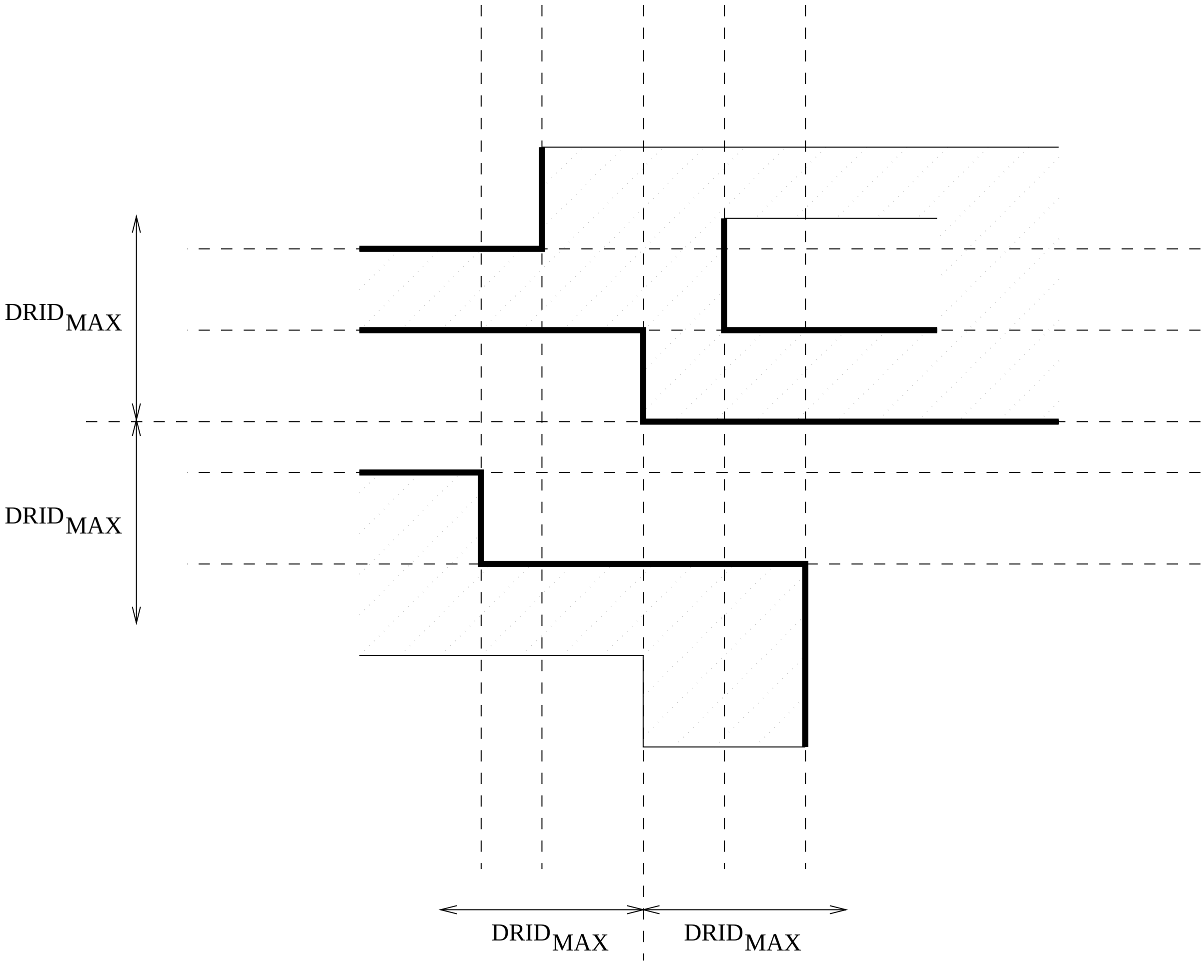


Figure 2.13: Intersecting vertical and horizontal Windows

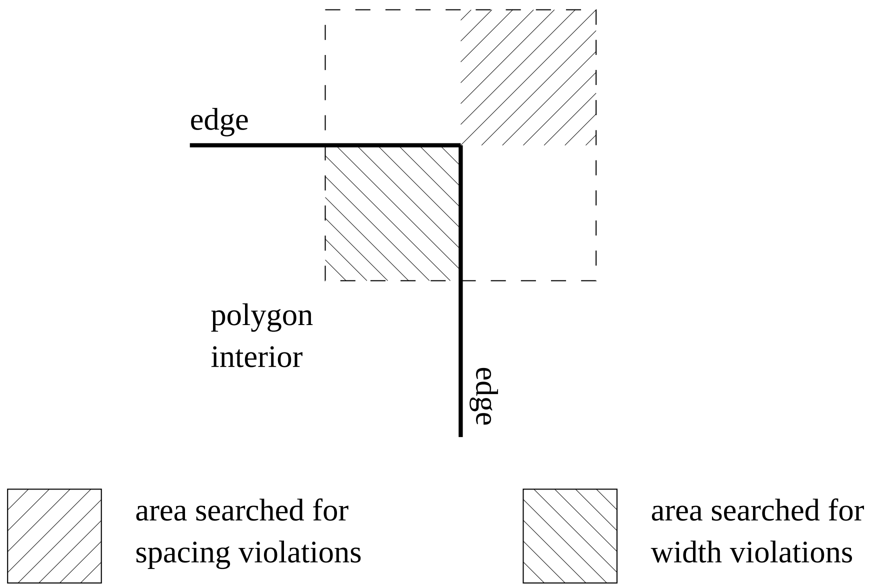


Figure 2.14: Regions searched by the ProperDRC width/spacing routine

Figure 2.15 shows the effect of using quadrants to limit the search space. The edge endpoint in the center of the window, indicated by the star, will be tested against edges 1, 2, and 3 for spacing violations, and against edges 4 and 5 for width violations.

Note that because the search areas in Figure 2.14 are larger than the ideal areas shown in Figure 2.11, some edges that fall within the search range may still be an acceptable distance away from the endpoint. Therefore, it is necessary to compute the distance to every edge that falls within the search range, to determine whether or not a clearance violation actually exists.

A trivial optimization to calculating the actual linear distance between two edges is to determine the square of the linear distance and compare this quantity with the square of the minimum width or spacing distance. The two advantages of using the squared

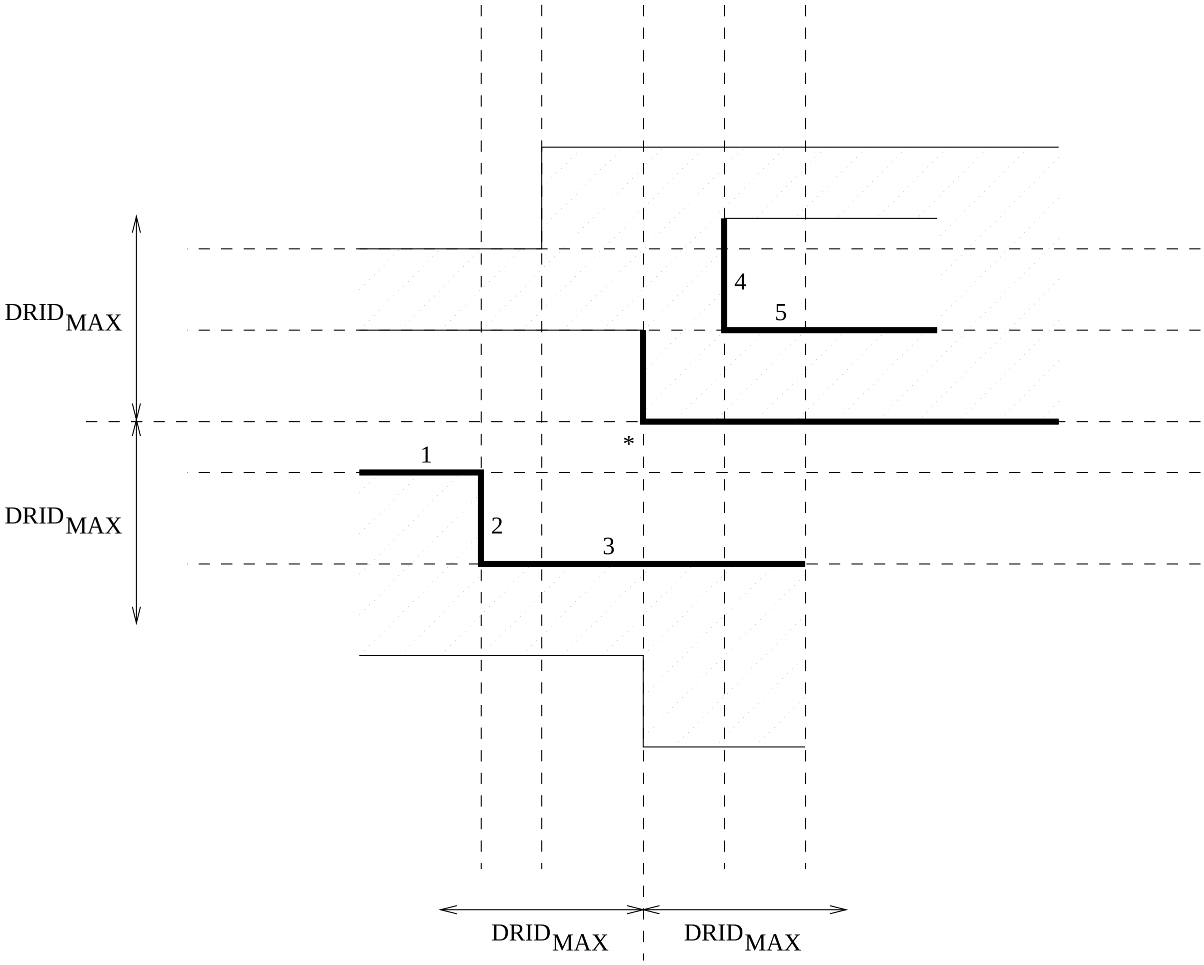


Figure 2.15: Edges to be tested for width and spacing violations

distance for comparison as opposed to using linear distance are that integer arithmetic may be used, and that the use of the relatively expensive square root function is avoided.

By restricting the edges tested for clearance violations to the ones that fall within a localized area, the ProperDRC width/spacing routine is able to achieve very good scalability. The ProperDRC width/spacing routine runs in $O(N)$ time for a set of N edges, which is faster than the Goalie algorithm developed by Szymanski and Van Wyk, while requiring a comparable amount of working space [8]. The derivation of the run time and memory requirements for the ProperDRC width/spacing routine is presented in Section 6.1.

3. PRIOR WORK IN PARALLEL DRC

Several approaches have been explored for parallelizing the design rule checking process in the past by other researchers. We will present an overview of previous work utilizing the following methods: area decomposition on flattened circuits, hierarchical decomposition on hierarchical circuits, functional decomposition on flattened and hierarchical circuits, and edge decomposition on flattened circuits.

3.1 Area Decomposition

Bier and Pleszkun have proposed a parallel algorithm that works on the flattened representation of mask layouts and uses an area decomposition strategy [9]. The circuit area is divided into subregions that are distributed to various processors, and each processor performs a complete set of design rule checks on its own subregion. The algorithm can work on polygon, pixelmap, or edge-based geometry representations.

Care must be taken when dividing the circuit area into subregions. A cut through the circuit area may introduce errors by dividing geometry features into pieces that do not pass the design rules by themselves. Furthermore, some design rule infractions may go undetected if the offending features lie on opposite sides of the dividing line. Both of these problems can be alleviated by extending the area of each processor's subregion on all sides by the maximum design rule interaction distance, which is defined to be the size of the largest constraint placed on the layout for a given technology. Any errors detected within the overlap region are discarded rather than reported. This technique is demonstrated in Figure 3.1.

This work did not specifically address the issue of load balancing. The circuit was partitioned by equal area regions. In chips with widely varying densities of rectangles, one region can have a large number of rectangles; hence, the speedup would be less than linear. We address this problem in our work.

A second problem is that the above work basically partitioned the chip area in a single dimension, by columns. It is well known that the perimeter of a square is less than that of a rectangle of equal area. Because the larger perimeter translates to an increase in the overlap area between processors, two-dimensional partitioning, as used in ProperDRC, minimizes the total amount of area assigned to each processor.

3.2 Hierarchical Decomposition

Unlike a flattened VLSI layout representation, in which all of the geometry features of a circuit are explicitly specified at all of the mask layers, a hierarchical representation of a VLSI layout groups sets of geometries into a single symbol, which usually represents

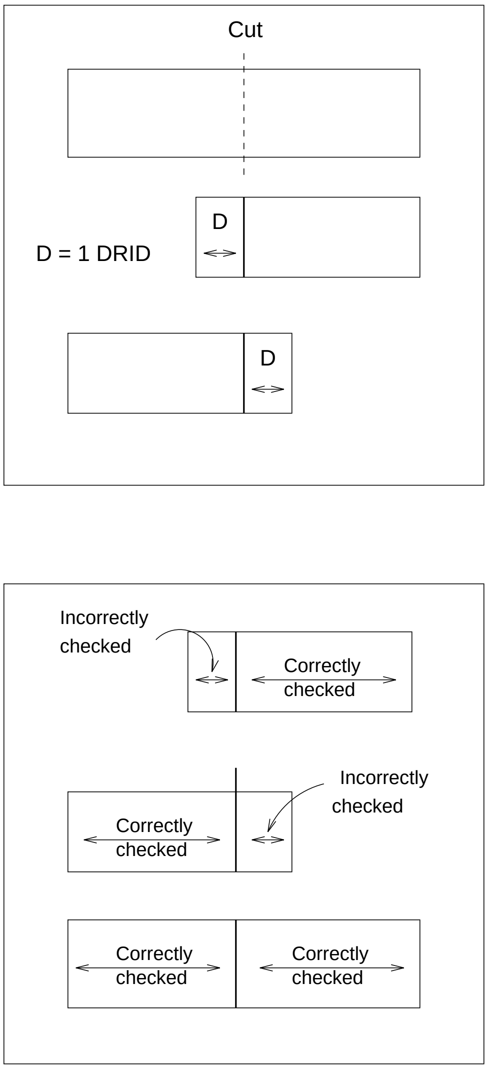


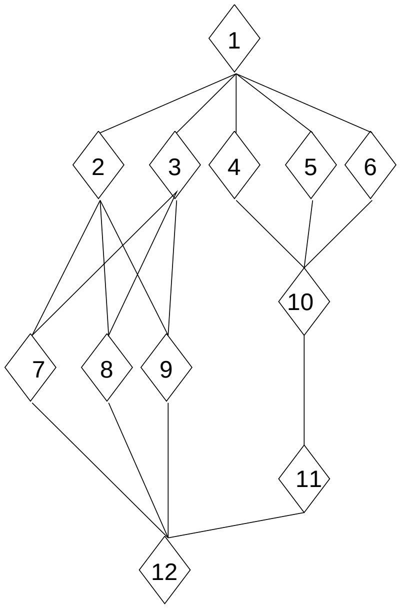
Figure 3.1: Overlapping subregions for area-based decomposition

a single functional unit of some type. Symbol calls can be nested, providing a tool for structured design.

A parallel DRC tool has been developed by Gregoretti and Seagall that takes advantage of the hierarchical representation of the circuit [10]. A generalized data type, called the token, is introduced to represent either a single geometry feature or a collection of features grouped into a symbol. The design rule checks are performed on the tokens themselves. When two tokens overlap, new tasks are generated in which one token is tested against all of the tokens represented by the second token, if it represents a symbol, or against the single feature the second token represents. The process is parallelized by having all processors take tasks from, and add tasks to, a common task queue.

This approach exploits parallelism only at the level of cells. If there are fewer cells in the design than processors in the multiprocessor, or if the cells have widely different sizes, there can be a load balancing problem. Also, this approach is not applicable to flattened circuit descriptions because there will only be a single task.

The other disadvantage with this approach is that it is not memory scalable. If the edge-based representation of a circuit is too large to fit in the memory of a single processor, the multiprocessor will not be able to operate on the circuit.



Heuristic	Height				Size			
Processor	A	B	C	D	A	B	C	D
time = 1	1				1			
time = 2	2	4	5	6	2	3	4	5
time = 3	3	10			6	7	8	9
time = 4	7	8	9	11	10			
time = 5	12				11			
time = 6					12			

Figure 3.2: DRC data dependency graph and its mapping on four processors

3.3 Functional Partitioning

Functional partitioning of the DRC process relies on the fact that a design rule check does not entail the execution of a single algorithm, but instead requires the sequential execution of many computationally independent algorithms. The goal of functional partitioning is to perform the computations necessary for separate rule checks simultaneously on different processors, while at the same time not duplicate the computations that contribute to the checking of more than one rule. Figure 3.2 shows an example data dependency graph containing 12 nodes representing DRC operations and its scheduling on four processors using two different heuristics.

Marantz developed a system that provided a general method of controlling the execution of any program that can be divided into a finite set of tasks [11]. This system was applied to the DRC problem by distributing the design rules to the various processors, where each processor applies its subset of rules to the entire circuit area.

The problem with this approach is that there is not enough functional parallelism available in realistic technologies. Therefore, the approach is not scalable. This method of parallelization also suffers from the same memory scalability problem as the previous approach, in that each processor must have enough memory to perform operations on the entire circuit area.

3.4 Edge Partitioning

Carlson and Rutenbar have developed an algorithm in which all scanline stops are generated at the start of the checking and then processed in parallel [12] and [13]. It is necessary to decompose the circuit geometry into a completely intersected set of edges, as shown in Figure 3.3, so that the set of all edges crossing a given scanline is immediately available. Boolean operations between layers, the determination of electrically connected sets of geometries, and checking for width, spacing, and extension violations are all performed in parallel on the scanlines.

This approach is applicable only to a single type of architecture, namely SIMD data parallel computers. This algorithm, therefore, is not appropriate for many of the powerful parallel machines available today.

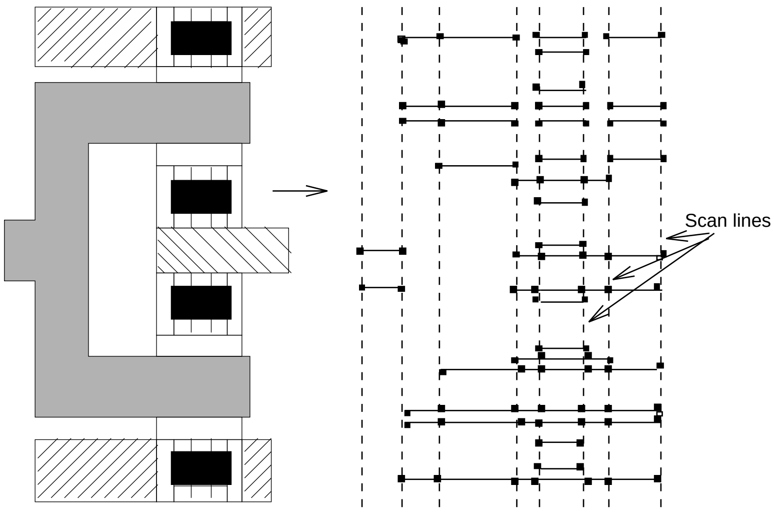


Figure 3.3: Completely intersected set of edges

4. A NEW APPROACH TO PARALLEL DRC

The serial design rule checking algorithm presented in Chapter 2 can be parallelized in two ways: First, the circuit area may be divided, and design rule checks performed on the subregions simultaneously; second, the series of elementary DRC tasks necessary to perform the checks for the various rules can be divided between processors.

These two methods of parallelizing the design rule checking process are completely independent of one another. Therefore, the data and task parallelism can be considered orthogonal axes of parallelism, in which exploitation of one of the two types of parallelism, or both simultaneously, will result in performance gains.

4.1 Data Parallelism

In ProperDRC, data parallelism is achieved by dividing the circuit in two dimensions into subregions and distributing the subdivisions of the circuit geometry between processors, or clusters of processors, depending on whether or not task parallelism is being implemented simultaneously. For the purposes of discussion in this section, let us consider the issues of implementing data parallelism by itself. Task parallelism, and a combination of the two types of parallelism, will be discussed in later sections.

The data partitioning scheme used in ProperDRC uses the number of rectangles assigned to a given processor as an estimate of workload. It should be noted that the actual amount of computation performed by a processor depends on the exact DRC checks performed on the rectangles within a region (see Chapter 6 for analysis of computations for various checks taking between $O(N)$ and $O(N \, log \, N)$ time for N rectangles). Partitioning the circuit by assigning equal area regions to each processor does not necessarily produce a balanced load, because the distribution of geometry features within the circuit area may not be uniform.

Several researchers have worked in the area of load balancing and partitioning [14]. Salmon [15] has proposed the use of the Orthogonal Recursive Bisection (ORB) scheme for solving the N-body problem [16] and [17]. Cybenko has reported on a scheme for recursive decomposition of workload in a multiprocessor [18]. Belkhale and Banerjee have proposed an alternate recursive partitioning algorithm for partitioning a set of points on a multiprocessor [19], and have reported implementations of this scheme in the context of a parallel circuit extractor [20] and [21]. All of the above partitioning methods are fairly complex to implement efficiently.

ProperDRC utilizes a data partitioning strategy based on a scheme proposed by Ramkumar and Banerjee [22] for parallel circuit extraction. The decomposition is performed by repeatedly subdividing the circuit area to produce subregions of equal area.

```
Procedure PARTITION:

  WHILE (not DONE) DO
      Receive batch of rectangles
      FOREACH rectangle in batch
          INSERT_ELEMENT(rectangle,mypartition)
      END FOREACH
  END WHILE

End Procedure


Procedure INSERT_ELEMENT(new_rectangle,this_partition):

  IF this_partition is a leaf node THEN
      Add new_rectangle to this_partition
      IF ++count_rectangles == grainsize THEN
          Create new equal area partitions left_part and right_part
          FOREACH rectangle in this_partition
              INSERT_ELEMENT(rectangle,this_partition)
          END FOREACH
      ENDIF
  ELSE
      IF new_rectangle overlaps left_part area
          THEN INSERT_ELEMENT(new_rectangle,left_part)
      IF new_rectangle overlaps right_part area
          THEN INSERT_ELEMENT(new_rectangle,right_part)
  ENDIF

End Procedure
```

Figure 4.1: Data partitioning algorithm

The subdivision continues until all processors have equal areas of the circuit geometry, and may continue further to facilitate load balancing. Figure 4.1 shows the pseudocode for the data partitioning algorithm.

The physical layout description is read from offline storage in the Caltech Intermediate Form (CIF) representation [23]. Rectangles are distributed to the corresponding processors in batches, so that it is never necessary to keep the entire circuit description in the memory of a single processor. The multiprocessor architecture can therefore operate on a circuit area that is too complex to fit into the memory of an individual processor.

The capability to perform layout verification on a circuit area that is too large for a uniprocessor is one of the most important advantages of performing design rule checking in parallel. The drawback associated with this method is that, because the entire circuit is never in the memory of a single processor at one time, no global quantification of the distribution of geometry features within the circuit area is possible. For this reason, circuit partitioning must be based purely on circuit area rather than dividing geometry features themselves equally among processors.

To balance the load between processors, additional decomposition is performed to further subdivide the chip area. A *grainsize* is specified by the user to limit the amount of additional decomposition performed. All areas that contain an amount of geometry features greater than the specified grainsize are subdivided. Circuit geometry regions are then remapped to processors in such a way as to provide the best load balancing.

Figure 4.2 shows the initial circuit partitioning on four processors for a sample circuit area, in which the X's represent geometry features. The dashed lines show the initial division of the circuit into equal-area subregions, which are assigned one per processor.

Figure 4.3 shows the same circuit after the load balancing algorithm is applied with a specified grainsize of 5. The region initially assigned to Processor 2 has been divided into two parts. Processor 3 will perform the DRC checks on the subregion on the right, to maintain better overall load balance.

Note that if a grainsize of 10 had been specified by the user, no further subdivision of the circuit beyond the initial partitioning shown in Figure 4.2 would have been performed.

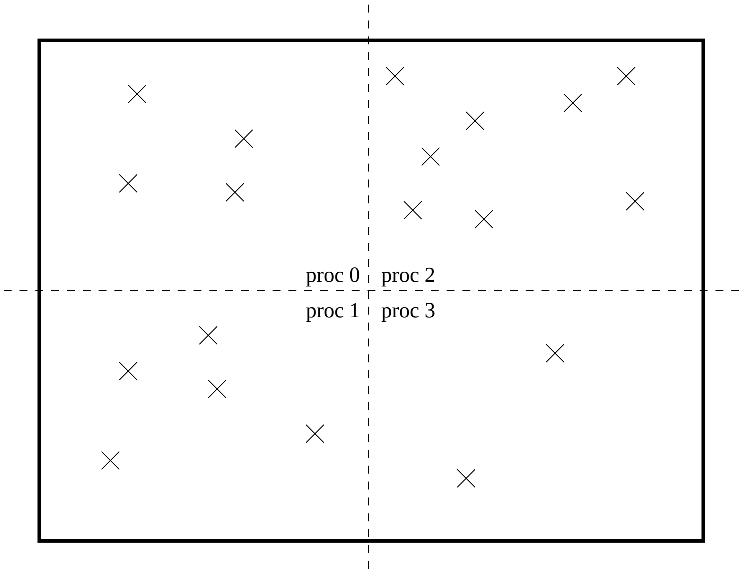


Figure 4.2: Equal-area circuit partitioning on four processors

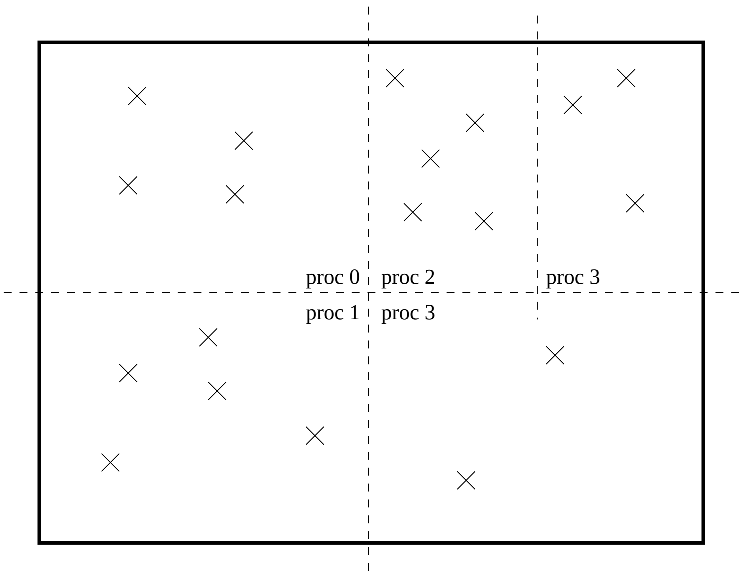


Figure 4.3: Circuit area redistribution with grainsize=5

The extent to which load balancing takes place is therefore completely under the user's control, which gives the user the flexibility to customize the performance of the algorithm to take full advantage of the multiprocessor architecture by selecting an appropriate grainsize.

The geometry layers are distributed in the original polygon representation used by the CIF input file. The conversion from polygon to edge-based representation takes place at the clusters that will perform the DRC tests on the area. Delaying the conversion until after the partitioning has two advantages: the messages are smaller, because one rectangle expands to two edges, and the conversion work is distributed to reduce the amount of time required.

Some overlap is necessary between the areas assigned to the various processors to ensure that no pairs of neighboring edges are overlooked. Each processor will receive all rectangles that lie within its area extended on all sides by the maximum design rule interaction distance for the technology. Figure 4.4 shows the partitioned circuit area from Figure 4.3 with the addition of the overlap areas. Rectangles that are present in more than one processor area will be duplicated and trimmed to the respective processor areas.

Trimming the rectangles can easily introduce geometry features that do not pass the design rules. The DRC routine must be careful not to report erroneous results introduced by the circuit partitioning. Therefore, upon completion of the design rule tests, infractions that fall within the maximum design rule interaction distance boundary

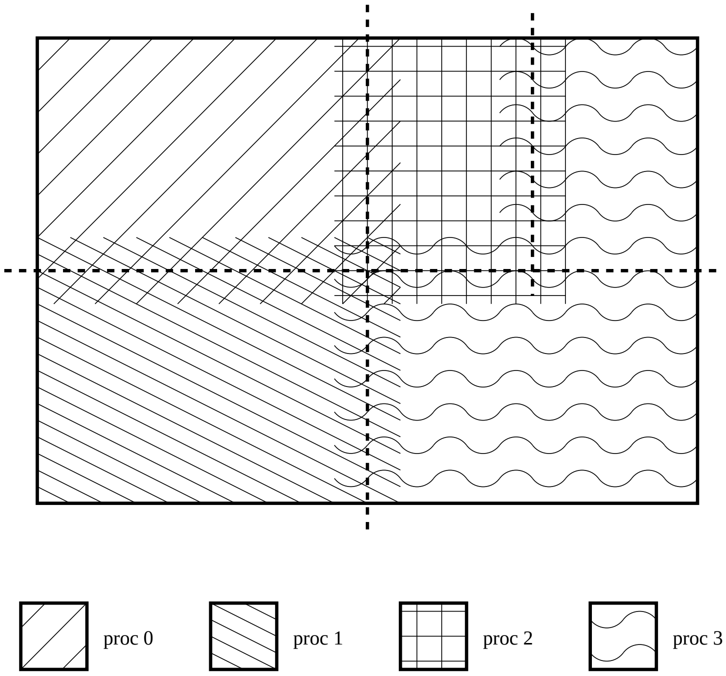


Figure 4.4: Overlapping processor areas

surrounding the processor's area of the circuit are disregarded rather than reported. This is similar to the approach outlined in Figure 3.1.

4.2 Task Parallelism

Task parallelism is achieved by having a group of processors share a single region of the circuit area and divide among themselves the elementary tasks necessary to perform the various DRC tests. If pure task parallelism is desired, the group of processors will actually be the entire set of processors in the multiprocessor architecture, each of which will divide up the DRC tasks for the whole circuit area. When a combination of data and task parallelism is used, the group of processors will represent an individual processor cluster inside the multiprocessor. Let us refer to the group of processors sharing the

DRC tasks for a given region of the circuit as a cluster, without loss of generality, for the purpose of the following discussion.

The task graph generated by the serial DRC algorithm is usable for the parallel DRC as well. The ideal parallel implementation would dynamically schedule the tasks, when upon the generation of a layer, all of the subsequent tasks that utilize that layer would be spawned on currently idle processors. However, such an implementation is not feasible, due to the dependencies in the task graph.

Elementary DRC operations such as the Boolean mask operations and Width/Spacing tests described in the previous chapter may have two input layers. These two layers will be generated by other tasks, which precede the current task in the task graph. It is conceivable (maybe even desirable, from a performance standpoint) that the two parent tasks run on separate processors in the cluster. The layers generated by both these parent tasks must be sent to a single processor, so that the subsequent task can be completed. Therefore, it is necessary that the destination processor for the generated layers, and thus the child task itself, be determined *a priori*. Other solutions to the dependency problem, such as broadcasting the EdgeSets or having the child task explicitly request the layer from the parents, introduce too much communication overhead to be effective.

The mapping of tasks onto processors is obtained by levelizing the task graph. Priorities are assigned to tasks based on the number of levels of subsequent tasks that depend on the output layers. Such priorities result in a scheduling that is comparable to using the

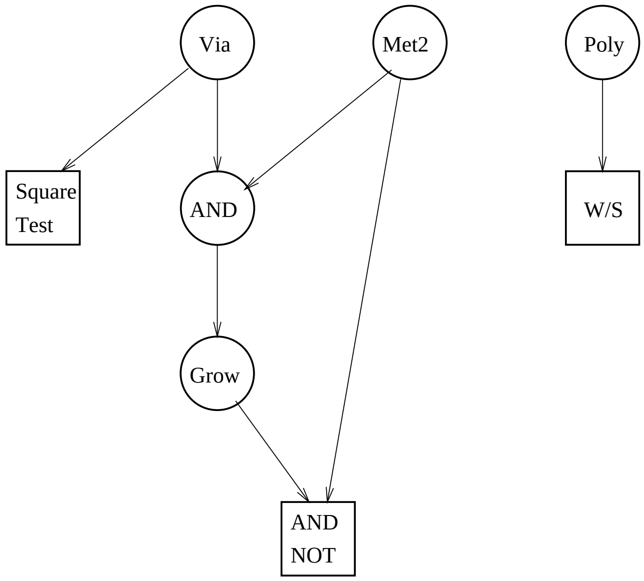


Figure 4.5: Sample task graph

height heuristic for mapping DRC functions onto processors, as described in Section 3.3. The levelized task graph is filled by arranging tasks in prioritized order.

For example, consider the sample task graph shown in Figure 4.5. The three tasks at the top of the graph represent the paint operation on the via, metal2, and polysilicon layers. The tasks shown in the graph correspond to a Square Test on the via layer, an Enclosure check on the via and metal2 layers, and a Width/Spacing test on the poly layer.

Figure 4.6 shows the task graph with each task tagged with a priority. In this example, a lower number represents a higher priority task. Priorities are assigned to the tasks beginning with the terminal layers and working up the graph. Each parent task is assigned a higher priority than its child task. The highest priority is used for tasks with more than one child, such as the paint tasks for the via and metal2 layers in the example.

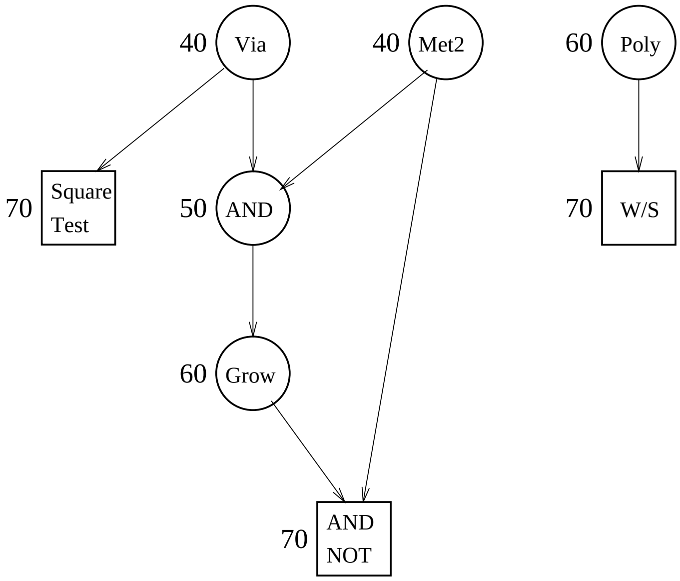


Figure 4.6: Graph of tasks with priorities

Figure 4.7 shows how the prioritized task graph can be mapped to a cluster that contains two processors. At each level, the highest priority tasks whose input layers have been generated are executed. Keeping parent and child tasks on the same processor, when possible, reduces the amount of communication necessary between the processors.

A task can begin as soon as its input layers arrive at the destination processor. There is no need for explicit synchronization between all of the processors of the cluster at the task graph level boundaries, so the penalty for load imbalance is not as severe as with the traditional barrier-synchronized implementation of a levelized task graph. Furthermore, because the complexities of the various algorithms used to implement the elementary tasks, as described in the previous chapter, can be used to estimate the length of time required to perform the operations for a given problem size, the potential exists for some intelligent scheduling methods to minimize the imbalance between processors.

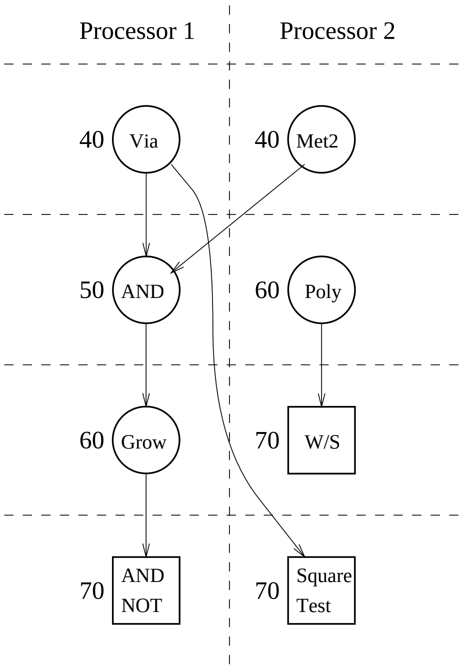


Figure 4.7: Mapping tasks onto a cluster

Because the number of tasks necessary to perform the design rule checks is fixed for a given technology, and is independent of the problem size, there is an upper bound on the performance that can be achieved by parallelizing these tasks, no matter how effective the load balancing strategies are. This would suggest task parallelism alone is not sufficient to obtain the best performance on a multiprocessor architecture with more than a few processors; a combination of data and task parallelism must be used.

4.3 Combination of Task and Data Parallelism

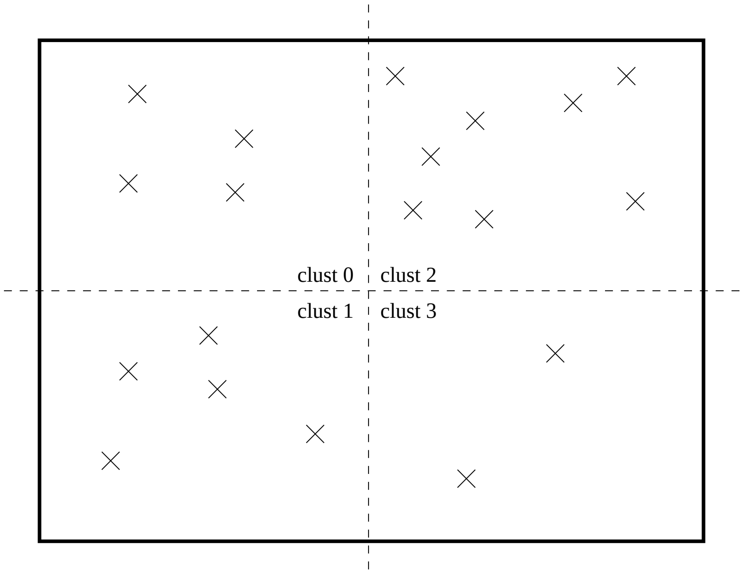
In the case in which a combination of data and functional parallelism is used, clusters of processors are assigned regions of the circuit area, and processors within the cluster perform the DRC tasks in parallel. Two separate load balancing issues must be addressed:

The elementary DRC tasks must be divided equally between the processors in each cluster, as discussed in the previous section, and the load must be balanced between the various clusters.

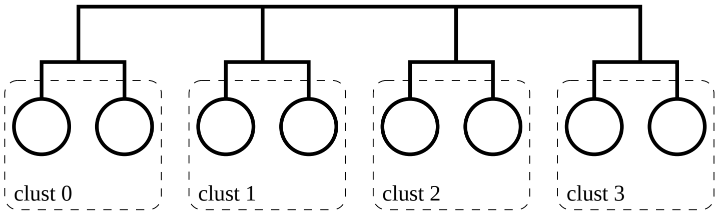
To balance the workload between processor clusters, a separate strategy is introduced. The same initial partitioning method is used as in the purely data parallel version, but the partitioning is done at the cluster level, rather than the processor level. The cluster estimates its own relative need for processing power, based on the number of geometry features inside its circuit region as a fraction of the total number of geometry features in the circuit. A fraction of the total number of available processors is then assigned to the cluster, based on this ratio.

The methods used to select which processors are apportioned to a given cluster can be customized to take advantage of physical locality in a given processor architecture. The end result is that a cluster with a lower workload "loans" one of the processors in its cluster to an overburdened cluster. In this way, more resources are applied to the more dense regions of the circuit area to improve the overall execution time.

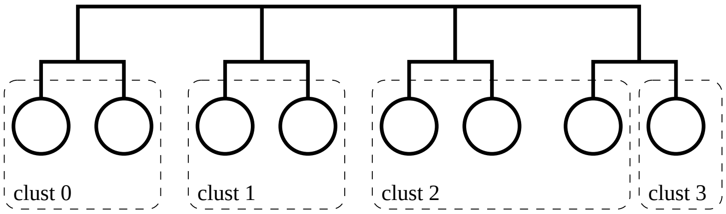
Figure 4.8 shows an example of how the load balancing scheme is applied, on an imaginary multiprocessor architecture with eight processors, arranged as four clusters of two processors. The partitioning of the circuit area between the clusters is shown in Figure 4.8(a). In the absence of the load balancing scheme, these circuit regions are assigned to each of the four homogeneous clusters, as depicted in Figure 4.8(b), in which the circles represent individual processors and the lines connecting them show the structure of the architecture. Figure 4.8(c) shows the processor-to-cluster mapping after application of the load balancing scheme. Cluster 3 has essentially borrowed an extra processor from Cluster 2 to compensate for the larger number of geometry features in its region of the circuit area.



(a) Partitioned circuit



(b) Original cluster mapping



(c) Balanced cluster mapping

Figure 4.8: Remapping processors to obtain balanced load between clusters

5. RESULTS

ProperDRC was used to test for violations of the MOSIS Scalable CMOS design rules [24]. A total of 32 design rules were specified, as presented in Appendix A, which resulted in the generation of 64 intermediate layers to perform all of the necessary tests. The following platforms were used to generate performance measurements: A Sun MP/670 shared-memory multiprocessor, a Sun SPARCserver 1000 shared-memory multiprocessor, a network of six Sun SPARCstations, and the CM-5 message-passing distributed-memory multiprocessor.

The benchmarks used to test ProperDRC include *plapart*, a programmable logic array with 25,000 rectangles; *kovariks*, a multiplier array with 64,000 rectangles; and *haab1* and *haab2*, static RAMs containing 128,000 and 253,000 rectangles, respectively. An artificial benchmark, *superhaab*, was also created, which consists of the *haab2* benchmark replicated four times, in array of two cells by two cells, with 10 λ spacing between cells. *Superhaab* contains 1,014,000 rectangles.

Tables 5.1 through 5.4 show the performance data for purely data parallel decomposition of the DRC. Dashes in the tables indicate that the processor configuration had insufficient memory to perform the DRC on the given circuit. The fact that the CM-5 was unable to operate on the *haab1*, *haab2*, and *superhaab* circuits with less than 8, 16, and 64 processors, respectively, illustrates the memory scalability of the ProperDRC algorithm.

It is also interesting that every platform appears to exhibit superlinear speedups as the number of processors increases from one to two. This is especially apparent with the larger benchmarks running on the Sun SPARCserver 1000, which run six to seven times faster on two processors than on a uniprocessor. This effect is most likely due to cache effects, where the smaller working space requirement of the two-processor implementation results in significantly fewer expensive memory operations.

Table 5.1: Data parallel performance on Sun MP/670 shared-memory multiprocessor

Circuit	Processors		
	1	2	4
plapart	326.5	111.0	47.5
kovariks	566.0	234.0	101.1
haab1	–	–	–
haab2	–	–	–
superhaab	–	–	–

The performance results for the purely task parallel implementation of ProperDRC are given in Tables 5.5 through 5.8. A small number of processors provide good performance results, but the effectiveness of adding additional processors diminishes quickly, for any

Table 5.2: Data parallel performance on a network of Sun SPARCstations

Circuit	Processors		
	1	2	4
plapart	175.1	59.5	28.9
kovariks	324.7	131.4	67.6
haab1	–	–	–
haab2	–	–	–
superhaab	–	–	–

Table 5.3: Data parallel performance on Sun SPARCserver 1000 shared-memory multiprocessor

Circuit	Processors			
	1	2	4	8
plapart	130.3	43.6	21.4	9.4
kovariks	244.7	94.3	38.1	24.2
haab1	843.1	114.5	64.1	40.7
haab2	1221.3	275.8	176.2	100.9
superhaab	–	–	–	–

Table 5.4: Data parallel performance on Thinking Machines CM-5 message-passing distributed-memory multiprocessor

Circuit	Processors							
	1	2	4	8	16	32	64	128
plapart	854.5	77.6	39.3	19.8	9.7	5.6	5.2	3.3
kovariks	–	288.2	89.9	45.0	24.6	12.0	6.9	6.2
haab1	–	–	–	261.5	100.3	44.7	34.8	29.5
haab2	–	–	–	–	159.8	128.9	59.2	69.3
superhaab	–	–	–	–	–	–	621.2	440.5

problem size. This is because the amount of task parallelism available is dependent only on the size of the set of technology rules being used, and not the size of the input file, as discussed in Section 4.2.

Table 5.5: Task parallel performance on Sun MP/670 shared-memory multiprocessor

Circuit	Processors			
	1	2	3	4
plapart	326.5	185.5	183.9	140.0
kovariks	566.0	321.3	274.8	242.5
haab1	–	–	–	–
haab2	–	–	–	–
superhaab	–	–	–	–

Table 5.6: Task parallel performance on a network of Sun SPARCstations

Circuit	Processors					
	1	2	3	4	5	6
plapart	175.1	104.4	100.0	90.4	94.5	76.2
kovariks	324.7	267.8	227.6	217.2	174.0	200.2
haab1	–	–	–	–	–	–
haab2	–	–	–	–	–	–
superhaab	–	–	–	–	–	–

Tables 5.9 and 5.10 show the performance results using a combination of data and task parallelism. It is important to notice that there are cases in which a combination of data and task parallelism provides better performance over either type of parallelism individually. A detailed analysis of these results is presented in the following chapter.

Table 5.7: Task parallel performance on Sun SPARCserver 1000 shared-memory multiprocessor

Circuit	Processors							
	1	2	3	4	5	6	7	8
plapart	130.3	76.1	65.7	56.8	53.5	53.5	44.7	42.1
kovariks	244.7	137.2	100.2	100.1	85.0	77.1	69.6	64.7
haab1	843.1	479.4	339.1	338.0	354.3	324.4	310.2	294.5
haab2	1221.3	683.5	575.6	502.8	476.9	457.6	419.1	422.8
superhaab	–	–	–	–	–	–	–	–

Table 5.8: Task parallel performance on Thinking Machines CM-5 message-passing distributed-memory multiprocessor

Circuit	Processors							
	1	2	3	4	5	6	7	8
plapart	854.5	442.6	350.6	311.7	306.2	280.5	233.1	308.7
kovariks	–	–	–	644.7	536.9	470.8	466.6	392.8
haab1	–	–	–	–	–	–	–	–
haab2	–	–	–	–	–	–	–	–
superhaab	–	–	–	–	–	–	–	–

Table 5.9: Data and task parallel performance on Sun SPARCserver 1000 shared-memory multiprocessor

Circuit	Processors/Clusters			
	4/2	6/2	8/2	8/4
plapart	28.9	25.5	21.9	14.3
kovariks	54.7	44.1	41.9	23.7
haab1	513.5	379.9	391.9	264.2
haab2	178.9	124.8	137.0	72.1
superhaab	–	–	–	–

Table 5.10: Data and task parallel performance on Thinking Machines CM-5 message-passing distributed-memory multiprocessor

Circuit	Processors/Clusters						
	4/2	6/2	8/2	8/4	12/4	16/4	16/8
plapart	157.3	127.1	108.1	59.6	46.1	19.7	26.0
kovariks	320.7	255.9	233.6	124.8	93.7	45.1	70.7
haab1	–	–	746.2	373.3	431.3	134.1	77.8
haab2	–	–	–	–	–	483.2	325.4
superhaab	–	–	–	–	–	–	–
Circuit	Processors/Clusters						
	24/8	32/8	32/16	48/16	64/16	64/32	128/64
plapart	20.3	17.2	12.6	10.2	9.8	8.7	10.7
kovariks	52.9	49.8	30.7	23.0	21.9	15.6	13.4
haab1	58.4	50.4	56.2	46.8	42.5	20.7	21.9
haab2	241.5	203.6	102.5	76.5	67.8	65.5	48.0
superhaab	–	–	–	–	–	326.7	301.2

The user-specified grainsize controls the extent to which load balancing takes place in the purely data parallel decomposition of the DRC problem. Any region of the circuit having a number of geometry features greater than the grainsize is subdivided into equal area regions, which may later be reassigned to different processors as necessary to facilitate load balancing. A lower grainsize will therefore result in a more even load distribution.

The effect of varying the grainsize for the purely data parallel decomposition is shown in Tables 5.11 and 5.12. The purely area-based circuit partitioning may be considered a degenerate case of the data partitioning strategy presented in this thesis, in which the grainsize is an infinite value.

Table 5.11: Effect of grainsize on data parallel performance on Sun SPARCserver 1000 shared-memory multiprocessor

Circuit	Grainsize	Processors			
		1	2	4	8
haab1	∞	843.1	311.1	111.2	57.0
	5000	843.1	124.6	65.3	41.1
	2000	843.1	114.5	64.1	40.7
haab2	∞	1221.3	911.7	462.6	197.5
	10000	1221.3	302.2	194.2	105.5
	5000	1221.3	275.8	176.2	100.9

Table 5.12: Effect of grainsize on data parallel performance on Thinking Machines CM-5 message-passing distributed-memory multiprocessor

Circuit	Grainsize	Processors				
		8	16	32	64	128
haab1	∞	395.3	239.9	83.1	36.3	29.5
	5000	–	100.3	79.2	34.8	30.4
	2000	261.5	104.2	44.7	45.7	30.3
haab2	∞	–	426.1	297.4	100.5	71.4
	10000	–	139.7	146.3	97.7	71.3
	5000	–	159.8	128.9	59.2	69.3
superhaab	∞	–	–	–	638.7	445.8
	50000	–	–	–	642.5	440.7
	20000	–	–	–	621.2	440.5

The concept of using task priorities to determine the order of execution for a set of DRC tasks was introduced in Section 4.2. Tasks are assigned higher priorities based on the number of levels of subsequent tasks that rely on the output of the task. Using an arbitrary ordering for tasks could result in a task schedule that produces more traffic and requires more waiting time than the prioritized schedule.

Table 5.13 shows the effect of using the priorities to schedule tasks, as opposed to using random ordering. These performance figures were taken on the network of Sun workstations, but the same concept is just as applicable to any of the platforms. The choice of a task ordering heuristic has no effect on the uniprocessor performance, as expected, because network traffic and processor idle time are not relevant concerns for uniprocessor execution. With two or more processors, the performance data illustrate that the prioritized task queue provides better performance.

Table 5.13: Effect of task scheduling on a network of Sun SPARCstations

Circuit	Task Ordering	Processors				
		1	2	3	4	5
plapart	random	175.1	142.4	116.4	109.7	92.2
	prioritized	176.0	104.4	100.0	90.4	94.5
kovariks	random	324.7	321.5	264.8	227.9	212.6
	prioritized	326.3	267.8	227.6	217.2	174.0

A cluster remapping strategy was presented in Section 4.3 as a means of balancing the load between clusters when a combination of data and task parallelism is used. Ideally, the number of processors assigned to a given cluster is proportional to the number of geometry features inside that cluster's region of the circuit area. However, because the total number of processors is fixed, and sometimes small, the fraction of the available processors assigned to a cluster cannot always equal the exact fraction of the total number of geometry features that lie within the circuit area owned by the cluster. Having a larger number of processors available allows the fraction of the available processors assigned to the cluster to more closely approximate the fraction of geometry features in the cluster area and, therefore, facilitates more effective load balancing.

Tables 5.14 and 5.15 show the effectiveness of the cluster remapping strategy. The load balancing had little effect on the Sun SPARCserver. This is to be expected, however, because the number of processors is relatively small, and it would take a very high imbalance to cause any of the processors to be remapped to a new cluster. The load balancing strategy was much more effective on the CM-5, where the large number of processors allows the processor-to-cluster mapping to be a bit more flexible.

Table 5.14: Effect of variable cluster size on Sun SPARCserver 1000 shared-memory multiprocessor for data and task parallel decomposition

Circuit	Cluster Size	Processors/Clusters			
		4/2	6/2	8/2	8/4
haab1	fixed	178.2	126.4	136.4	63.9
	variable	178.9	124.8	137.0	72.1
haab2	fixed	515.0	382.5	394.0	268.1
	variable	513.5	379.9	391.9	264.2

Table 5.15: Effect of variable cluster size on Thinking Machines CM-5 message-passing distributed-memory multiprocessor for data and task parallel decomposition

Circuit	Cluster Size	Processors/Clusters							
		16/8	24/8	32/8	32/16	48/16	64/16	64/32	128/64
haab1	fixed	78.2	58.4	56.1	72.6	50.7	52.4	25.8	22.1
	variable	77.8	58.4	50.4	56.2	46.8	42.5	20.7	21.9
haab2	fixed	388.9	260.8	237.5	123.7	92.5	85.3	87.5	62.9
	variable	322.8	241.5	203.6	102.5	76.5	67.8	65.5	48.0
superhaab	fixed	–	–	–	–	–	–	335.2	362.4
	variable	–	–	–	–	–	–	326.7	301.2

6. ANALYSIS OF APPROACHES

To analyze the performance results of ProperDRC, we will first examine the performance of the serial algorithms used to implement the various DRC operations. We will then proceed to examine the performance issues introduced by the parallelization of the DRC process.

6.1 Analysis of Serial DRC

The complexities of some of the established DRC algorithms have been discussed in Section 2.4. ProperDRC uses the scanline operation developed by Lauther to perform Boolean mask operations [7], and the edge sorting algorithm presented by Szymanski and Van Wyk [8]. We will now analyze the performance of the width/spacing clearance checking algorithm presented in this thesis.

Let N represent the number of edges in the layer(s) to be tested for width/spacing violations. Let S denote the number of different X-coordinates in a set of edges (i.e., the number of scanline stops) and M denote the maximum number of edges cut by any vertical line (i.e., the maximum number of edges on a given scanline). Lauther asserts that the expected value of both S and M is $O(\sqrt{N})$ [7]. The following arguments show that the width/spacing clearance checking algorithm takes $O(N)$ time and requires $O(M)$ space.

There will be S scanline stops, as the vertical scanline passes from left to right over the circuit. At each scanline stop, the vertical window surrounding the current scanline must be generated to include all scanlines within the maximum design rule interaction distance ($DRID_{MAX}$), which is constant for a given technology. Let the value W denote the $DRID_{MAX}$ divided by the minimum feature size for the technology. W is therefore a constant, and the number of scanlines the vertical window may contain is bounded by $2 * W + 1$.

For each scanline added to the vertical window, the vertical edges on the scanline must be generated. These are generated by sequentially stepping through the horizontal edges that cross the scanline, so the time taken is $O(M)$. Because the number of scanlines in the window is bounded by a constant, the amount of time taken to generate the entire window is $O(M)$.

Once the vertical window has been generated, the horizontal window is passed down the length of the vertical window. The number of horizontal windows generated for a given vertical window is bounded by M. As Figure 6.1 demonstrates, the number of distinct edge endpoints that the intersection between the vertical and horizontal windows may contain is bounded by $(2 * W + 1) * (2 * W + 1)$, which is a constant. The endpoint

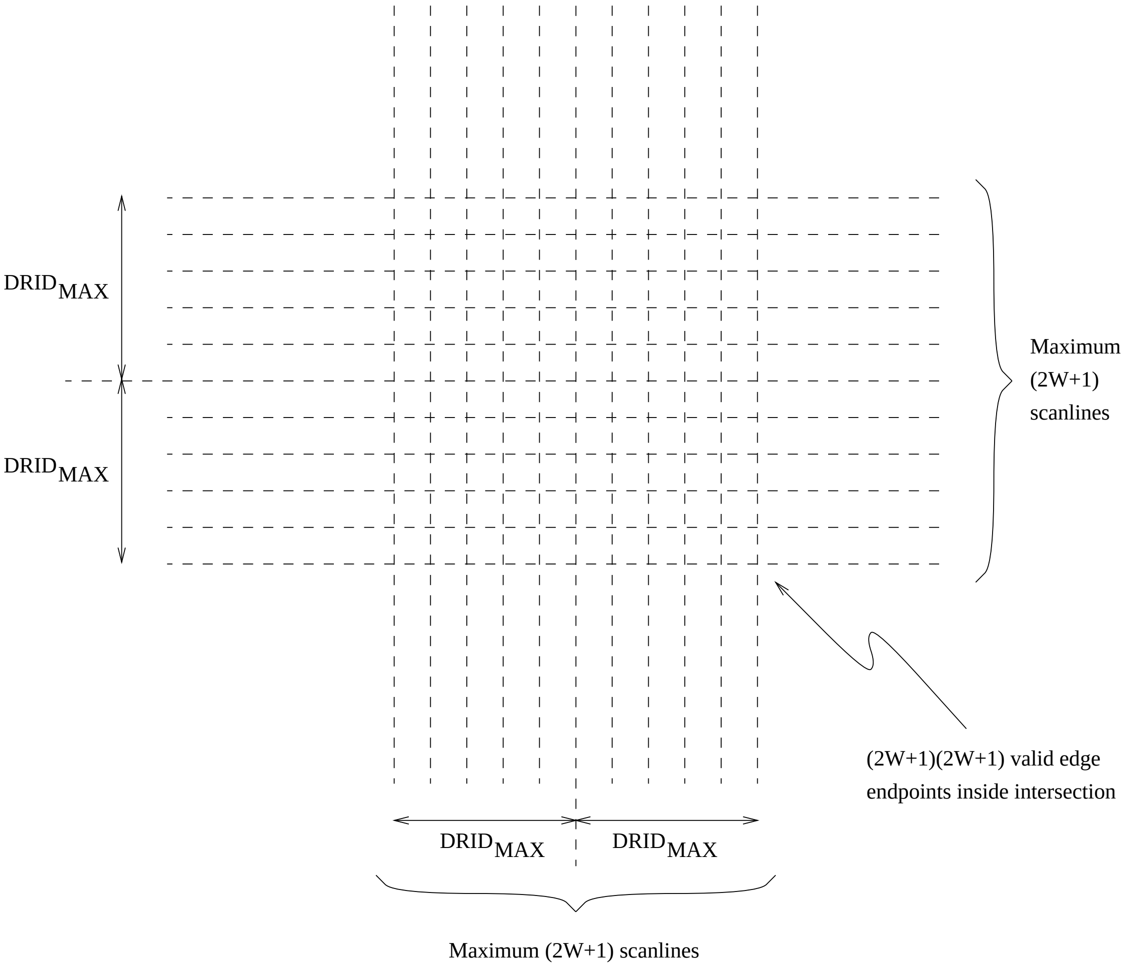


Figure 6.1: Maximum number of edge endpoints inside the window intersection

at the center of the horizontal window is therefore tested against a constant number of edges, which takes a constant amount of time. The time required to add and delete edges from the horizontal window is also bounded by a constant. Therefore, the amount of time to generate and perform DRC checks on all of the horizontal windows corresponding to a given vertical window is bounded by $M * O(1) = O(M)$.

At each scanline stop, the vertical window must be generated, and the horizontal windows within that vertical window must be generated and checked. So for S scanline stops, the time required is $S * [O(M) + O(M)]$. Using Lauther's assertion on the bounds for S and M, the total time taken for the width/spacing check is $O(N)$.

Only one vertical window is in memory at a given time. The number of scanlines in the window is bounded by a constant, and there are no more than M edges per scanline; therefore, the amount of memory required by this algorithm is $O(M)$.

Table 6.1 provides a summary of the complexities of the various algorithms used to perform the DRC operations. Note that there is an implicit Sort of cost $O(N\ log\ N)$ associated with the first scanline operation on the edges of a given layer (namely, the *paint* operation). Subsequent sorting operations on the same layer can take advantage of the natural output order of the scanline operation to perform the sort in $O(log\ N)$ time.

Table 6.1: Complexities of the DRC operations

Operation	Execution Time
Boolean Operation	$O(N\ log\ N)$
Sort	$O(N\ log\ N)$ or $O(log\ N)$
Grow	$O(N\ log\ N)$
Width	$O(N)$
Spacing	$O(N)$
Square Test	$O(N)$
Overall DRC	$O(N\ log\ N)$

An exception to this savings in sort time is the sort performed on the layer produced by a Grow operation. Because the edge endpoints are moved in different directions depending on whether they correspond to interior or exterior polygon vertices during the Grow operation, the natural ordering of the scanline output is disrupted and, therefore, the Sort following a Grow operation will take $O(N\ log\ N)$ time. Considering that the overall performance of the DRC is bounded by the performance of the most complex algorithms used by the DRC, the overall complexity for the DRC is $O(N\ log\ N)$.

6.2 Analysis of Parallel DRC

The performance results demonstrate that both data parallelism and task parallelism can be applied to the DRC problem to achieve better performance and reduced memory requirements as compared to serial algorithms. Because neither of the two types of parallelism adversely impacts the effectiveness of the other, a combination of the two types of parallelism can be applied to achieve further parallelism.

In practice, the ultimate goal is to achieve the best performance given an existing architecture. Table 6.2 shows some of the performance results measured on the CM-5 from the previous chapter, rearranged to show a comparison between using pure data parallelism and using a combination of data and task parallelism on a given number of processors. In the case of the combination of data and task parallelism, the number of processors per cluster given in the table is an average value; the processor-to-cluster mappings may be modified to balance the load between clusters, as discussed in Section 4.3.

Table 6.2: Comparison of parallelization methods on CM-5

Circuit	Procs per Cluster	Processors			
		16	32	64	128
haab1	1	100.3	44.7	34.8	29.5
	2	77.8	56.2	20.7	21.9
haab2	1	159.8	128.9	59.2	69.3
	2	325.4	102.5	65.5	48.0

The data in Table 6.2 show that neither of the two parallelization approaches is superior to the other in all cases. There are two counteracting factors that affect the relative performance of the two algorithms: The task parallel performance is limited by the complexity of the DRC algorithms, and the data parallel performance is limited by the extra work created by the overlapping processor areas.

To illustrate the effect of the complexity of the DRC algorithms on the task parallel performance, consider a simplified case in which two processors are to be applied to perform a design rule check on a circuit with $2X$ geometry features. Assume perfect load balancing, whether data or task parallelism is used.

No matter which type of parallelism is used, the combination of the two processors must have the memory capacity to hold the entire circuit. Using the complexity of the slowest algorithms in the DRC procedure, the time necessary to perform the DRC on a circuit of problem size N is $O(N\, log\, N)$, and the minimum amount of working space required for the DRC algorithms is $O(\sqrt{N})$. We will use the term *working space* to distinguish between the amount of storage required by a single processor to perform the various DRC operations, and the amount of storage required by the entire set of processors performing the DRC operations to hold the whole circuit geometry, which is fixed at $O(N)$ for the entire set of processors.

If data parallelism were used to divide the circuit's geometry features equally between the processors, neglecting the overlapping processor areas for the moment, each of the processors would perform a DRC on a subregion of the circuit with X geometry features.

The total run time would be $O(X\, log\, X)$, because this amount of time is necessary for each processor to perform local design rule checking simultaneously. The demand for work space at each of the processors is $O(\sqrt{X})$.

In the task parallel implementation, the various DRC tasks would be divided equally between the two processors. Both processors would be working on a problem size of $2X$. The time required for the DRC would be $O(1/2 * (2X)\, log\,(2X)) = O(X\, log\,(2X))$. The working space requirement for each processor would be $O(\sqrt{2X})$. Therefore, the task parallel version of the DRC requires slightly more time to run and more working space. These penalties are $O(constant)$, but nonetheless indicate that the purely data parallel implementation provides the better performance when overlapping processor areas are disregarded.

Now, let us take the effect of overlapping processors into consideration. Figure 6.2 shows a square circuit area with area A. Using two-dimensional partitioning to divide the circuit area between P processors, the resulting circuit region assigned to each processor is a square whose side measures $\sqrt{A}/\sqrt{P}$. Taking the overlap area of c units on every side of the processor area into consideration, the total area assigned to the processor is $(\sqrt{A}/\sqrt{P} + 2c) * (\sqrt{A}/\sqrt{P} + 2c)$, or $A/P + 4c\sqrt{A}/\sqrt{P} + 4c^2$ units. This area formula can be generalized to $A/P + kc\sqrt{A}/\sqrt{P} + 4c^2$ for decompositions of circuit areas that result in processor areas that are not perfectly square, where k is a constant.

Note that the actual area assigned to processors whose regions are on the outside boundaries of the circuit is actually slightly lower. These slight area discrepancies can be

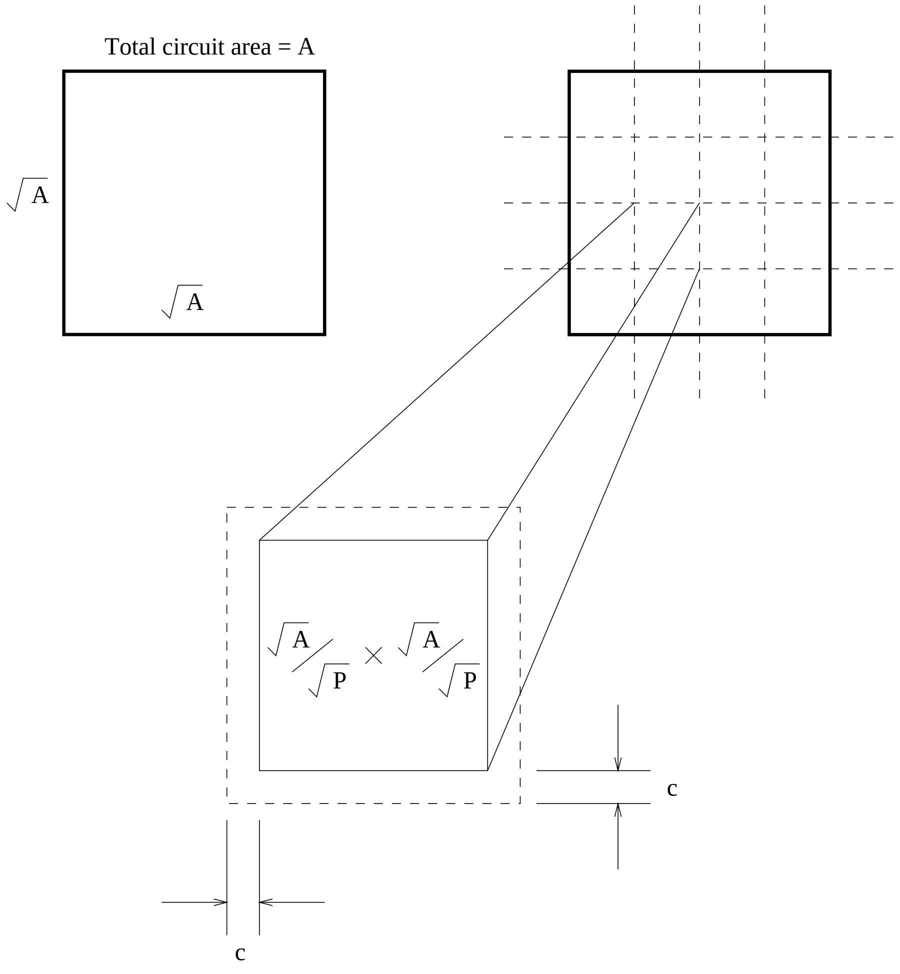


Figure 6.2: Dividing the circuit area between processors

safely ignored, because a smaller fraction of the processors is on the boundary as the total number of processors increases, and the performance of the algorithm on the circuit as a whole will be bounded by the processors with the highest areas. Disregarding these area discrepancies, the total area operated on by the set of P processors is $A + kc\sqrt{AP} + 4c^2P$.

Returning to our fictitious example of dividing a circuit with $2X$ geometry features between a pair of processors, the actual amount of area assigned to each of the two processors is $X + kc\sqrt{2X} + 8c^2$. The actual amount of time consumed by the slowest of the DRC algorithms is now $O((X + kc\sqrt{2X} + 8c^2)\, log\,(X + kc\sqrt{2X} + 8c^2))$, and the working space requirement is $O(\sqrt{X + kc\sqrt{2X} + 8c^2})$. Note that these penalties are dependent on the number of processors.

In addition to the penalty for the complexity of the DRC algorithms associated with the task parallelism, there is also the overhead of intraprocessor communication, whereas the purely data parallel decomposition of the problem requires no communication *while the DRC checks are being performed*, although some communication is necessary during the data partitioning phase to perform the load balancing between processors.

The experimental results also show that load balancing is more difficult to attain for the task parallel implementation as compared to the data parallel implementation. Consider that the task parallel DRC on the *plapart* benchmark on the network of Sun workstations took longer with five processors than with either four or six. The actual distribution of the geometries between the different mask layers is much more critical for the task parallel implementation than for the data parallel version. The particular distribution in the *plapart* benchmark apparently presented a load balancing problem for the particular order in which the layer operations were divided between five processors.

The biggest impediment to the effectiveness of task parallelism is the complexity of the DRC algorithms. If improvements in the existing algorithms, such as the new width/spacing clearance checking algorithm presented in this thesis, can reduce the execution times to $O(N)$ or better, task parallelism may become an attractive alternative to pure data parallelism. The development of a load balancing scheme that can compensate for variations in the distribution of the geometries between the mask layers will also be important in achieving better performance with task parallelism.

7. CONCLUSIONS

ProperDRC has several merits. It includes an original algorithm for geometry clearance checking that outperforms current published algorithms. It can accommodate a wide variety of changes in technology without recompiling. It is capable of achieving parallelism by using data parallelism, task parallelism, and a combination of the two types.

A number of areas in parallel design rule checking should be explored in the future. Ideally, a DRC tool should be able to exploit the hierarchy of large designs. Performing DRC on a flattened layout representation may result in much redundant work if individual cells in the design are instantiated a large number of times, as is often the case with library cell-based designs.

ProperDRC could also be expanded to handle non-Manhattan layout geometries. Most of the algorithms used in ProperDRC would require a minimal amount of additional work to be capable of operating on non-Manhattan designs. The arguments presented in the discussion of the time and working space requirements of the original width/spacing test in this thesis are equally valid for designs with non-Manhattan features. The Grow algorithm would have to be modified, because an orthogonal grow is probably not sufficient for reliable preservation of non-Manhattan geometries.

The ProperCAD programming model has several advantages over other parallel programming environments. The strict adherence to the C++ language allows easy integration of existing code with new code. Incremental parallelism allows the development of the serial algorithm to be unencumbered by parallelization issues. The Actor model serves as an abstraction of the actual architecture and, therefore, supports implementation on such disparate architectures as shared-memory and message-passing multiprocessor systems. The resulting CAD tool is usable on a variety of multiprocessing platforms.

APPENDIX A. SAMPLE DRC TECHNOLOGY FILE

The technology file is used to specify design rules for a given process technology. The technology files presented in this appendix use SCMOS layer names and lambda rules to express feature sizes. Lambda rules do not necessarily have to be used; feature size measurements could be expressed in microns or any other distance units.

Integers are used to refer to layers in the technology file. Table A.1 lists the SCMOS layer names and the corresponding integer assigned to the layer. Layer numbers 2 and 3 are used by ProperDRC to allow separate rules to be specified for p-diffusion and n-diffusion layers. Because SCMOS technology does not differentiate between the p-diffusion and n-diffusion layers, layer 3 is unused for this technology.

Table A.2 shows the formats of the various rule specification statements used in the technology file. A single letter denotes one of the seven basic rules that ProperDRC is capable of testing for violations.

Some example rule specifications are presented in Figure A.1. For this sample file, a selection of rules dealing with the via layer is presented. An asterisk in the first column

Table A.1: SCMOS layers

Number	Name	Description
1	CPG	polysilicon
2	CAA	active area
3	–	not used
4	CWP	p-well
5	CMF	metal1
6	CMS	metal2
7	CCP	metal-poly contact
8	CCA	metal-diffusion contact
9	CVA	via

Table A.2: Format for design rule specifications

Rule specification	Description
W (layer) (width) (spacing)	single-layer Width/Spacing
S (layer1) (layer2) (spacing)	two-layer Spacing
E (layer1) (layer2) (enclosure)	Enclosure
O (layer1) (layer2)	Overlap
Q (layer) (size)	Square Test
N (layer1) (layer2)	No-Overlap
X (layer1) (layer2) (extension)	Extension

```
*
*  Some sample rules:
*
Q 9   3               VW          via size
W 9   3   4           VS          via spacing
S 9   1   2           VSG,VSI     via spacing to poly
E 9   6   2           SOV         metal2 overlap of via
O 9   6               SOV         metal2 overlap of via
```

Figure A.1: Excerpts from a sample technology file

of a line in the technology file represents a comment. These lines are not parsed as rules. The ability to insert comments into the technology file allows for a more readable format.

The first rule specifies that all features in the via layer are squares, with dimension of 3x3. The second statement enforces a minimum distance of 4 λ between vias. The third rule allows a minimum separation of 2 units between features in the via and polysilicon layers. The fourth rule specifies that the metal2 layer must surround a via by a minimum distance of 2 λ on all sides. The final rule specifies that a via feature must have a metal2 geometry feature overlapping it.

It may not seem apparent why it is necessary to differentiate between the last two rules specified in the example technology file. It turns out that there are a few cases in which the former rule must be enforced, but the latter need not apply. An example is a minimum distance by which the p-well layer must enclose an active area region. The Enclosure rule must be used to specify this minimum distance. However, not all active areas must reside in p-wells, because an active area might also appear inside an n-well. Therefore, the Overlap rule would not apply in this case.

In the sample technology file, a verbal description of the rule appears on the same line as the statement defining the rule. The description of the rule can be printed along with other diagnostic information when an infraction of the rule is encountered. This results in a legible output format for the DRC.

The entire technology file used to implement the MOSIS design rules is presented in Figure A.2. These design rules expand to the task graph shown in Figure A.3. In the task graph, the circles represent intermediate layers, which are used to compute subsequent layers. The squares represent terminal layers, which represent the result of one of the DRC tests. The edges in the terminal layers are the actual edges in the circuit geometry that fail to meet meet the design criteria.

```
*
* poly rules
*
W 1   2   3           GW,IW,GS,IS  poly width, spacing
X 1   2   2           GOA,ISA      poly gate extension, spacing to active area
E 7   1   2           IOC          field poly overlap of contact
*
* via rules
*
Q 9   3               VW           via size
W 9   3   4           VS           via spacing
E 9   2   2           VSA          via spacing to active area
O 9   2               VSA          via spacing to active area
S 9   1   2           VSG,VSI      via spacing to poly
S 9   7   3           VSC          via spacing to contact
S 9   8   3           VSC          via spacing to contact
N 9   7               VOC          via overlap of contact
N 9   8               VOC          via overlap of contact
*
* metal1 rules
*
W 5   3   3           FW,FS        metal1 width, spacing
E 7   5   2           FOC          metal1 overlap of contact
O 7   5               FOC          metal1 overlap of contact
E 8   5   2           FOC          metal1 overlap of contact
O 8   5               FOC          metal1 overlap of contact
*
* contact rules
*
Q 7   2               CW           contact size
Q 8   2               CW           contact size
W 7   2   3           CS           contact spacing
W 8   2   3           CS           contact spacing
S 7   8   3           CS           contact spacing
S 7   1   3           CSG          contact spacing to gate poly
S 8   1   3           CSG          contact spacing to gate poly
*
* metal2 rules
*
W 6   4   3           SW,SS        metal2 width, spacing
E 9   6   2           SOV          metal2 overlap of via
O 9   6               SOV          metal2 overlap of via
*
* pwell fules
*
W 4   6   6           WpW, WpSs    pwell width, spacing
E 2   4   3           WpOAp        pwell overlap of active area
*
* active area rules
*
W 2   6   4           AW, ASA      active area width, spacing
E 8   2   2           AOC          active area overlap of contact
O 8   2               AOC          active area overlap of contact
```

Figure A.2: Technology file for MOSIS design rules

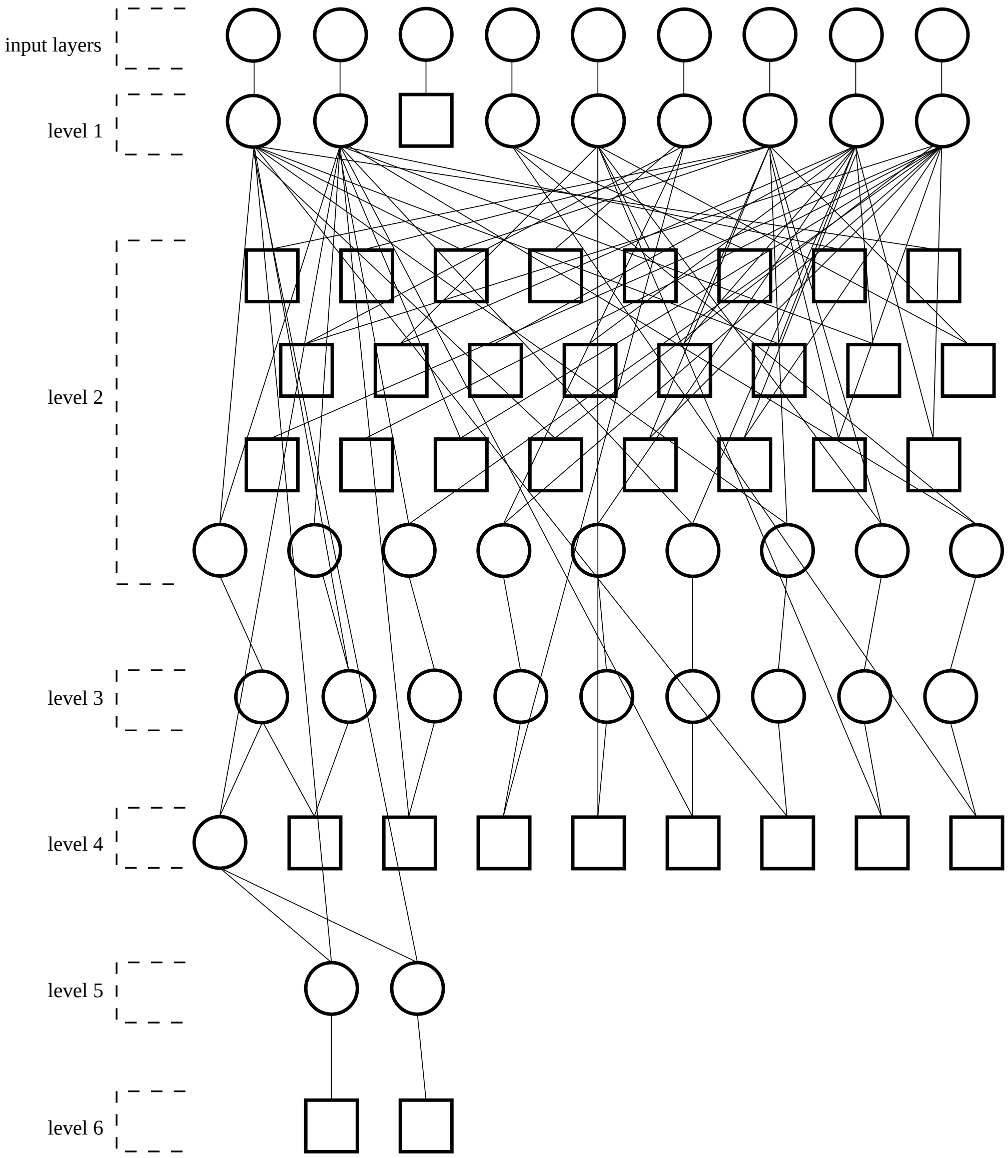


Figure A.3: Task graph for MOSIS design rules

APPENDIX B. SAMPLE INPUT FILE

The circuit geometry is specified in the CIF format. Figure B.1 shows an example input file that describes the layout of a CMOS inverter. This particular input file uses only the simplest constructs of the CIF language. ProperDRC also supports some of the more advanced capabilities of the CIF language, such as nested symbol definitions and transformations.

Table B.1 presents some of the commonly used CIF commands. The Layer command is used to specify the current mask layer. For SCMOS technology, the layer names presented in Figure A.1 are used. The Box command describes a polygon in the current layer, with the specified dimensions and center point (cx, cy).

The Definition Start and Definition Finish commands enclose the specification of the geometry of a cell in the design. The *cell* argument specifies a cell number, and the *ascale* and *bscale* arguments are used to scale distance measurements in the cell by a fixed amount, to produce a shorter and more legible cell definition. The Call Symbol command is used to instantiate the cell specified by the *cell* argument. The

```
DS 1 25 2;
9 inverter;
L CWP;
    B 624 504 -48 0;
L CMF;
    B 696 48 -12 240;
    B 48 72 180 180;
    B 48 108 -276 -198;
    B 708 48 -18 -276;
L CPG;
    B 144 36 -204 306;
    B 36 144 -150 216;
    B 36 180 18 234;
    B 132 36 -102 126;
    B 120 36 72 126;
    B 36 72 -54 72;
    B 36 36 114 90;
    B 204 36 -138 18;
    B 132 36 66 54;
    B 36 24 -222 -12;
    B 60 36 -210 -42;
    B 36 36 -198 -78;
    B 36 132 18 -30;
    B 252 36 -90 -114;
L CAA;
    B 504 384 -48 0;
L CCA;
    B 24 24 180 168;
    B 24 24 -276 -168;
DF;
C 1;
End
```

Figure B.1: Sample CIF input file

Table B.1: Common CIF commands

Name	Format
Layer	L *name* ;
Box	B *length width cx cy* ;
Definition Start	DS *cell ascale bscale* ;
Definition Finish	DF ;
Call Symbol	C *cell* [*transformation*] ;
User Extension	*integer text* ;

optional $transformation$ argument may specify any combination of translating, rotating, or mirroring along the x- or y-direction of the original cell geometry.

The User Extension commands allow the creator of the CIF file to provide additional information about the circuit. The sample CIF input file shown in Figure B.1 includes a User Extension command to associate the textual name "inverter" with cell number 1. User Extension commands can also be used to label specific geometry features of interest in the circuit, such as the locations of the input and output ports of a cell.

APPENDIX C. SAMPLE RUN

The command line arguments necessary for invoking ProperDRC are as follows:

`properDRC` *clusters filename grainsize*

The *clusters* argument is an integer that specifies the number of processor clusters to be used. This number must be a power of two. The *filename* argument is the name of the input file that describes the circuit geometry. The *grainsize* argument is used to select a load balancing strategy. A positive *grainsize* value will cause ProperDRC to implement the pure data parallel load balancing algorithm discussed in Section 4.1. A negative value selects the cluster size load balancing strategy discussed in Section 4.3. A zero *grainsize* will run the DRC with no load balancing.

Figure C.1 shows the output from a sample execution on a four-processor Sun MP. Two clusters and no load balancing are selected. The input file is the inverter from Figure B.1.

```
% properDRC 2 inverter.cif 0

Bounds: minx = -32, miny = -26, maxx = 29, maxy = 28
Numrects = 22

[0] 19(2) 72(3)
[1] 26(2) 51(4)
[2] 24(2) 72(3)
[3] 26(2) 51(4)


 Compute time:       0.25           Total time:       0.65

 num of clusters:       2
 procs per cluster:     2
 total num procs:       4

properDRC: done
```

Figure C.1: Output from sample ProperDRC run

The first two lines of output show the bounds for the layout area, converted to the lambda units used by the MOSIS design rules, and the number of geometry features in the circuit.

The next four lines are a list of design rule infractions discovered in the circuit. The number in square brackets is the processor number that found the infraction. Following each processor number is a list of terminal layers and the number of violations of that particular design rule. This brief output format holds the output to a reasonable size for any number of errors. Some of the benchmarks used to test ProperDRC generate thousands of DRC violations, so this brief format keeps the DRC output manageable.

However, a much more detailed error report can be produced for the benefit of the user. ProperDRC keeps track of the exact location of the offending edges, as well as the verbal description of the design rule provided in the technology file. The user can easily configure ProperDRC to print a detailed error report, so that the errors in the circuit geometry can be easily corrected.

The last lines in the output show the performance statistics. The *compute time* is the amount of time taken to perform all of the DRC checks, in seconds. The *total time* also includes the I/O time necessary to read the description of the circuit from offline storage and distribute the data to the processor clusters, and print the results upon completion of the DRC.

REFERENCES

[1] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design.* Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Trans. Comput.-Aided Des.*, vol. 13, pp. 829–842, July 1994.

[3] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel enrironment for VLSI CAD," Tech. Rep. CRHC-93-04/UILU-ENG-93-2205, University of Illinois, Coordinated Science Laboratory, Jan. 1993.

[4] S. Parkes, J. Chandy, and P. Banerjee, "ProperCAD II: A run-time library for portable, parallel, object-oriented programming with applications to VLSI CAD," Tech. Rep. CRHC-93-22/UILU-ENG-93-2250, University of Illinois, Coordinated Science Laboratory, Dec. 1993.

[5] S. Parkes, *A Class Library Approach to Concurrent Object-Oriented Programming With Applications to VLSI CAD.* Ph.D. dissertation, University of Illinois, Dept. of Elec. & Comp. Eng., 1994.

[6] B. Preas and M. Lorenzetti, eds., *Physical Design Automation of VLSI Systems.* Menlo Park, CA: Benjamin/Cummings Publishing, 1988.

[7] U. Lauther, "An O(N log N) algorithm for Boolean mask operations," in *Proc. 18th Design Automation Conf.*, pp. 555–562, June 1981.

[8] T. Szymanski and C. J. Van Wyk, "Goalie: A space efficient system for VLSI artwork analysis," *IEEE Design Test Computers*, vol. 2, pp. 64–72, June 1985.

[9] G. E. Bier and A. R. Pleszkun, "An algorithm for design rule checking on a multiprocessor," in *Proc. Design Automation Conf.*, pp. 299–303, June 1985.

[10] F. Gregoretti and Z. Segall, "Analysis and evaluation of VLSI design rule checking implementation in a multiprocessor," in *Proc. Int. Conf. Parallel Processing*, pp. 7–14, Aug. 1984.

[11] J. Marantz, "Exploiting parallelism in VLSI CAD," in *Proc. Int. Conf. Computer Design*, Oct. 1986.

[12] E. Carlson and R. Rutenbar, "Design and performance evaluation of new massively parallel VLSI mask verification algorithms in JIGSAW," in *Proc. 27th Design Automation Conf.*, pp. 253–259, June 1990.

[13] E. Carlson and R. Rutenbar, "Mask verification on the Connection Machine," in *Proc. Design Automation Conf.*, pp. 134–140, June 1988.

[14] S. H. Bokhari, "Partitioning problems in Parallel, Pipelined and Distributed computing," *IEEE Trans. Comput.*, vol. C-37, pp. 48–57, Jan. 1988.

[15] J. K. Salmon, *Parallel Hierarchical N-body Methods.* Ph.D. dissertation, California Institute of Technology, Dec. 1990.

[16] J. E. Barnes and P. Hut, "A hierarchical O(N log N) force calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[17] J. P. Singh et al., "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity," *Parallel & Distrib. Comput.*, vol. 27, pp. 118–141, June 1995.

[18] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Parallel & Distrib. Comput.*, vol. 7, pp. 279–301, July 1989.

[19] K. P. Belkhale and P. Banerjee, "Recursive partitions on multiprocessors," in *Proc. 5th Distributed Memory Computing Conf.*, Apr. 1990.

[20] K. P. Belkhale and P. Banerjee, "Parallel algorithms for VLSI circuit extraction," *IEEE Trans. Comput.-Aided Des.*, vol. 10, pp. 604–618, May 1991.

[21] K. P. Belkhale, *Parallel Algorithms for CAD with Applications to Circuit Extraction.* Ph.D. dissertation, University of Illinois, Dept. of Computer Science, 1991.

[22] B. Ramkumar and P. Banerjee, "ProperEXT: A portable parallel algorithm for VLSI circuit extraction," in *Proc. 7th International Parallel Processing Symposium*, pp. 434–438, 1993.

[23] C. Mead and L. Conway, *Introduction to VLSI Systems.* Philippines: Addison-Wesley, 1980.

[24] J.-I. Pi, *MOSIS Scalable CMOS Design Rules.* Information Sciences Institute, University of Southern California, Marina del Rey, CA.