Space-Efficient Extraction Algorithms

N.P. van der Meijs, A.J. van Genderen
Delft University of Technology
Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands

ABSTRACT
In this paper, we describe how we have limited the space complexity of a layout to circuit extractor by (1) a combination of the scanline technique with the corner stitching technique, (2) a region-based extraction algorithm, (3) a judicious choice of netlist format and (4) a union-find data structure also supporting deletions of elements. The efficiency of the new algorithms and the resulting extractor is confirmed by experimental data. These results are important, since in practice the size of the largest design that can be handled is often hard-limited by available memory.

1. Introduction
The ever-increasing complexity of VLSI circuits puts ever-increasing demands on layout analysis and verification software. Despite the success of the hierarchical design style, final verification is often done on a flattened layout. The problem then is how to organize and handle the geometric and netlist data such that layouts containing $10^6$ – $10^8$ and more rectangles/trapezoids can be handled. For each particular computer or workstation, the size of the largest design that can be handled is often hard-limited by available memory. This is also true in a virtual memory environment where thrashing can severely degrade performance. Thus, it is imperative that layout analysis programs be economical of computer memory—and based on memory-efficient algorithms.

The algorithmic technique that potentially requires the least memory is the scanline technique, see e.g. [1,2]. With scanline algorithms, only the geometry intersected by a moving scanline need to be in the core memory of the computer.

With layout to circuit extraction, however, this low space complexity is seldom realized. Often, the extracted circuit is entirely built-up in the core memory of the computer. Then, the $O(N)$ space complexity realizable by scanline algorithms is superseded by the $O(N)$ amount of memory needed to store the circuit. A notable exception is the Goalie system [3,4], where a low overall space complexity is achieved by using a union-find algorithm with backwards renaming of regions together with splitting the extraction process into a sequence of distinct programs for e.g. connectivity analysis, transistor identification and parameter calculation.

In this paper, we describe how we have reduced the space complexity of the SPACE layout to circuit extractor. Unlike the system in [3,4], this program performs all extraction operations in one single scanline sweep over the layout—the input is a layout and the output is the extracted circuit, including parasitic capacitances and resistances.

An almost $O(N)$ space complexity of SPACE has been achieved by a combination of the scanline technique with the corner stitching technique (to be described in Section 2), a region-based extraction algorithm (Section 3), a judicious choice of netlist format (Section 4) and an efficient union-find data structure also supporting deletions of elements (Section 5). Section 6 presents experimental data confirming the low space complexity of the algorithms and the resulting extractor. As was found in [3,4] also, reducing space complexity need not be at the cost of increased computation time. This is also confirmed in Section 6.

2. Scanline Based Region Enumeration
In this section we will briefly summarize the geometric algorithms of SPACE as originally described in [5]. SPACE employs a combination of the scanline paradigm, see for example [1,4], and the corner stitching technique [6] for representation and analysis of the layout. The scanline algorithm enumerates all tiles and pairs of adjacent tiles in a consistent order, well suited for circuit extraction as described in Section 3. The tiles are regions of maximal vertical dimension and of one color, where the color of a tile must be interpreted as a binary n-tuple or bitmask indicating the presence and absence of masks. To allow arbitrary angles, the tiles are trapezoidal instead of rectangular as in [6]. The upper and lower tile boundaries can have any angle, but the tiles have vertical left and right boundaries, one of which may be of zero length.

The scanline algorithm works on the non-vertical contour edges of each layer. This input is obtained from a polygon description of the layout by a separate program [7], and serves as the common data format for all layout verification and analysis programs at our site.

At the start of the algorithm, there is one tile covering an infinite plane as in Figure 1(a). This plane is then scanned from left to right. During this operation, the subdivision of the plane as a set of covering but non-overlapping tiles is refined. For example, when the left endpoint of an edge is encountered, we have discovered the lower-left corner of a new tile above the edge and the upper-left corner of the tile below the edge. This is illustrated in Figure 1(b). When a right endpoint of an edge is encountered, we have discovered the lower-right corner of the
tile above the edge and the upper-right corner of the tile below the edge as depicted in Figure 1(c). Note from Figure 1(d) that an edge can have an arbitrary number of horizontally abutting tiles above and below it.

Tiles can be linked together at their corners as in [6]. The links are called corner-stitches. They enable efficient inspection of the neighborhood of a tile or fast area searches such as “find all tiles within a certain distance to the left or below a given tile”. Each tile contains four stitches, two at the bottom-left corner (the bl and the lb stitch) and two at the upper-right corner (tr and rt). Slightly different from [6], the stitches are defined as follows. For each tile t:

- bl points to the bottom-most tile whose right edge is incident to t.
- tr points to the top-most tile whose left edge is incident to t.
- lb points to the left-most tile whose top edge shares a finite segment with the bottom edge of t.
- rt points to the right-most tile whose bottom edge shares a finite segment with the top edge of t.

This modification is necessary when a moving band of corner-stitched tiles is implemented. An illustration of a corner-stitched plane is given as Figure 2.

To reduce memory requirements, a moving band of corner-stitched tiles can be realized by maintaining a FIFO-queue of completed tiles. Each time a tile is completed, it is injected into the queue. Thus, the queue contains the tiles ordered according to their right abscissa, since this is the order in which the tiles are completed. Each time the scanline advances, the back of the queue is inspected to see which tiles leave the band so that their storage space can be reclaimed.

The connection of the region enumeration algorithm with the actual extraction algorithms is via three operations that are executed by the layout module and implemented by the circuit module. Here, we describe the external interface of these operations, their implementation is described in the next section.

**EnumPair** (tile1, tile2) This operation is executed for every pair of adjacent tiles, as found while scanning the layout.

**EnumTile** (tile) This operation is executed for each finished tile, that is, a tile of which the top-right corner has been processed by the scanline.

**ClearTile** (tile) This operation is executed for each tile that leaves the corner-stitching band (leaves the queue of tiles in this band), just prior to memory de-allocation.

The order in which these operations are executed is illustrated using Figure 3, where the numbers indicate the order in which the tiles are created. Pairs of two numbers indicate the EnumPair operation and single numbers indicate the EnumTile operation, the ClearTile operation is not shown.

### 3. Circuit extraction

All extraction operations (including e.g. device recognition, connectivity analysis and capacitance computation) are performed during one single sweep of the scanline over the layout. As illustrated in Figure 4, only that part of both the layout and the extracted circuit that is incident to the scanline, is...
determines what circuit elements are present. For example, when the color of the tile has the bits corresponding to the diffusion implant and polysilicon masks turned on, an n-enhancement transistor and a polysilicon conductor layer are identified.

Once transistors and conductor layers are known, nets and elements of the same type that are present in both tiles are merged. For example, conductor layers are connected, and adjacent transistor parts are combined into one transistor. As the transistors are combined, their dimensions Wre the source perimeter and the total perimeter, are also updated.

EnumPair also determines features such as edge capacitances from the color transitions between two adjacent tiles and coupling capacitances by following the corner stitches, as described in [5].

3.2 The EnumTile Operation

EnumTile receives one tile as an argument and handles circuit elements that are related to the surface of that tile. For example, it updates transistor surfaces and finds parallel plate capacitances between overlapping conductor layers. Also, when a contact is present in the tile, EnumTile merges the affected nets.

To find resistances, SPACE optionally uses a finite element technique that decomposes each tile into triangles and that associates a resistor with each edge of a triangle [9, 10]. These operations are also done in EnumTile.

3.3 The ClearTile Operation

To achieve low memory requirements, neither the layout nor the extracted netlist can remain in core until the extraction is finished. Instead, as soon as the connectivity of a net or circuit element has been resolved, the ClearTile operation will write that item to the disk and the core memory is reclaimed. As will be shown in Section 4, it is not necessary to keep a net in core until all other nets are known.

4. Netlist Format

There are 2 basic types of netlist data structures or data formats. We have found that for layout to circuit extraction, a so-called net-based format requires less computer memory than a so-called instance-based format and is therefore to be preferred.

An instance-based format is basically an enumeration of all instances with their connections (net numbers) and parameters. An example of an instance-based format is a SPICE circuit description. A net-based format, on the other hand, is one in which the instances and the nets are stored in two separate lists. The first list enumerates the instances with their parameters but without connectivity information. This information is specified in the second list which enumerates the nets together with the (instance name, port name) pairs making up the net. An example of a net-based circuit format is EDIF [11, 12], where the instances are specified with the (instance ...) construct and the nets with the (net ...) construct.

With a net-based format, the instances and their connections are stored separately. Consequently, as soon as a device has been recognized it can be written out to the instance file and the core memory allocated for it can be freed. Then, only some (instance name, port name) pairs linked to the nets connecting to the device must remain in core, but only until the nets are finished. When this is the case, the (instance name, port name) pairs can be written out to the net file.

With an instance-based format, devices can only be written out after the nets connecting to them are completely known. Since many devices have at least one connection to a large net, (e.g. power, ground or clock) they must, on the average, stay in core longer than with a net-based format.

We conclude that an instance-based format will have significantly higher memory requirements then a net-based format. Consequently, we have implemented a net-based format in SPACE. In Section 6, we will present measurements that confirm the low number of nets and devices remaining in core memory.

5. Union find

In a scanline-based extraction program, connectivity analysis is complicated by structures as the one shown in Figure 5 (see also [4]) because only when the scanline reaches tile c, it is found that tiles a and b are part of the same net. This problem is handled as follows: Each tile that contains interconnect initially inaugurates a new net. The EnumPair operation subsequently merges the nets of two adjacent interconnect tiles. When a tile later leaves the corner-stitching band, memory for it must be deallocated in ClearTile in order to reduce the amount of core memory. It are the deletions that cause trouble, as discussed below.

The connectivity analysis problem can be abstractly stated as the task of maintaining a collection of disjoint sets, in a dynamic situation where a sequence of the following operations

---

1. To facilitate the discussion, we assume without loss of generality that there is only one interconnect layer.
occurs in arbitrary order:

**Find**  Execute a query asking for some identification of the set to which an element belongs. Frequently, pairs of such queries are executed to determine whether or not two elements belong to the same set.

**NewSet**  Create a new singleton set containing a newly created element.

**Union**  Perform the union operation on two sets, and return the new set. The two old sets are destroyed.

**Delete**  Delete an element.

A version of this problem without the Delete operation, is known as the union-find problem and an efficient yet simple algorithm that solves it is known [13,14]. We will briefly describe that algorithm below and subsequently extend it to allow Delete operations also. A more detailed description can be found in [15].

The union-find problem is solved using a data structure that is a forest of trees—one tree for each set and one tree node for each element. Each tree node, except the root, is linked to its parent in the tree. Sets are identified via the root of the tree. The Find operation traverses the path from the element to the root and returning the root, see Figure 6(a). The Union operation makes one tree a subtree of the other, see Figure 6(b).

The key to the efficiency of these operations is the addition of two heuristics: (1) path compression and (2) union by weight, see [13,14].

Non-leaf nodes cannot easily be deleted, since this would break the structure of the union-find tree. Maintaining extra links to repair the tree after deletion of a node does not lead to an efficient solution. Instead, we propose lazy deletion of non-leaf nodes. When a non-leaf node has to be deleted, it is retained in the tree but marked as being condemned. Only when it subsequently becomes a leaf, it is actually deleted.

With lazy deletion, the union-find tree will generally contain condemned nodes that should be deleted but nevertheless occupy memory. However, the two heuristics ensure efficiency because many non-leaf nodes will quickly become leaf nodes. When compared to the total number of nodes, the number of condemned nodes is low. This claim is confirmed by results in Section 6.

**6. Results**

In this section we will briefly present some measurements confirming the efficiency of the new algorithms and the resulting extractor. These results were obtained by flat extraction of a number of randomly selected layouts of varying size, on an HP-9000/720 workstation. In the figures that follow, each data point is for another design. Where solid lines are drawn, they do not indicate fittings. Rather, they are meant for easy visual comprehension.

Figure 7 confirms the practicality of a net-based netlist format: the maximum number of transistors in core memory is much lower than the total number of transistors.

Figure 8 confirms the efficiency of the union-find algorithm with lazy deletion. For each design, its maximum number of active nets (maximum of the number of nets that are intersected by the scanline over all scanline stops) is displayed on the x-
axis. Its maximum number of total nets in core memory (i.e., active + condemned) is displayed on the y-axis. The optimal ratio of total nets vs. active nets is 1 and is shown by the solid line. The results clearly indicate that only a small number of condemned nets, proportional to the number of active nets, remains in core memory awaiting actual de-allocation. Further analysis reveals [15] that in most cases, around 20% of all nets are condemned.

Figure 9 displays the amount of memory used vs. the size of the design measured in its transistor count. This figure clearly demonstrates a sublinear space complexity that is achieved by the combination of techniques described in this paper.

That reducing space complexity need not be at the cost of increased computation time, was also recognized in [3, 4]. This is confirmed by Figure 10, displaying computation time vs. transistor count. The solid line represents a $O(N)$ complexity of 100 transistors/second.

7. Conclusion

In this paper we have described a 4-fold way to reduce the space complexity of a layout to circuit extractor and Section 6 has presented measurements that confirm their practicality. The new algorithms are implemented in a tool called SPACE. Indeed, a designer can now without difficulty perform detailed extraction of large and relatively flat layouts on his own workstation, inside the design-loop.

Although Section 6 has presented data for flat extraction, we must note that SPACE in fact is a hierarchical extractor. However, in many cases flat extraction is inevitable (sometimes in the case of Sea-of-Gates designs or in the case of designs imported from other systems), or just more accurate.

SPACE is integrated in the NELSIS IC Design System [16]. It works comfortably together with other tools in the system such as a layout editor, a network comparison (graph isomorphism) tool and a switch-level simulator. The NELSIS system provides interfaces to, among other formats, the GDSII and CIF layout formats and the Spice and EDIF netlist formats.

References