Semi Greedy Algorithm for Finding Connectivity in Microchip Physical Layouts

Clemente Izurieta
VLSI Research and Development Laboratories
Hewlett Packard Co.
3404 E Harmony Rd. MS#32
Fort Collins, CO 80525
clemente.izurieta@hp.com

Abstract

Scan based or Line Sweep methods are a traditional mechanism to traverse the physical layout, or artwork of a microchip. These traversals are incremental in nature. They typically traverse the artwork from bottom to top (in the y direction) and from left to right (in the x direction) in a traditional Cartesian plane. As the traversal occurs, client applications need to know what causes a given event. For example, an analysis algorithm may be interested in finding out every time two rectangle shapes overlap or touch. This is important because such an overlap could represent a *short* in a circuit. The number of rectangles used to represent real world designs of artwork in a microchip is in the order of one million per metal layer, thus various optimization techniques are necessary to breakdown the complexity of such designs. This paper describes a semi-greedy technique coupled with an object oriented design pattern to aid in traversing and reporting events of chip data.

Introduction

CAD (Computer Aided Design) tools for customized chips have a much longer software engineering lifecycle than other products, with applications and architectures typically staying in place for large periods of time. This is in part due to the complexity and extended lifecycles of full custom CPU and microprocessor designs, but mostly to the large amount of dependencies between the various tools and associated methodologies involved. Such dependencies are common when derivative design methodologies are used. A derivative design methodology re-uses pre-existing rules from previous microchips making it hard to change individual components.

Over the last couple of years, however, we have seen an emergence of object oriented architectures and technologies appear within this space. This is prompting researchers to take an active role in investigating and contributing to new object oriented architectures and more efficient algorithms in the CAD field that facilitate faster and more flexible software lifecycles. The software lifecycle of chip design spans many domains, including behavioral, structural, and physical. We focus on the physical domain. Central to the physical domain are scan-based algorithms. They form one of the many core engines upon which various CAD analysis tools are built. This has been the motivating factor behind this investigation and paper. It is these core engines that will need to become more *"modifiable by design"* in order to facilitate faster turn around times in the lifecycles of analysis tools.

Scan-based algorithms are also known as line sweep algorithms in various literatures. They all try to solve the problem of finding segment and/or geometric figure intersections. In the context of this paper, the practical application is VLSI CAD tools. Various analysis tools make use of such traversal algorithms in order to find electrical connectivity, find rule violations (electrical or physical), discover routes across metal layers, etc. There are two key components of traversal algorithms. First, the algorithm must be efficient, especially when dealing with millions of geometric figures, and second, the algorithm must have an extensible mechanism to report events as they happen.

We describe a semi-greedy approach to performing the scan-based traversal. It is this approach that reduces the number of geometric figures to be considered at any given point and helps optimize the core traversal engine. As the traversal occurs, we also describe an event reporting mechanism based on the *command design pattern* [2].

Background

Greedy algorithms

Greedy algorithms are used to help solve optimization problems. They use all the best locally available data to make a decision. According to [1], Greedy algorithms have two properties: a) *The greedy choice property* – A globally optimal solution can be arrived at by making a locally optimal (greedy) choice, and b) *The optimal structure property* – A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems.

A typical greedy algorithm example is the *Activity Selection* problem. This is basically a problem of scheduling various tasks of variable lengths to use a single resource. The goal is to schedule the maximum possible number of tasks. Each task has a start time *S*i, and a finish time *F*i.

The algorithm's pseudo code (from [1]) is as follows:

```
. Set A ← {1}
. j ← 1
. For i = 2 to N
    do if Si >= Fj  then A← A U {i}
                         j = i
. Return A
```

Each activity is sorted in ascending finishing times. This algorithm can schedule a set of N activities in O (n) assuming *Fi* already sorts them.

Design Patterns

According to [2], design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." An important property of object-oriented design is encapsulation and reuse of data and algorithms. Taking this notion one step further, you find that many designs and implementations of more complex systems tend to exhibit a lot of similarities, thus, patterns begin to emerge that are generic in nature. The encapsulation of these patterns in such a way that they can be reused in different contexts is yet another level of abstraction. There exist behavioral, structural, creational, and many other types of patterns.

Semi-Greedy algorithm for finding connectivity in physical layouts

As previously explained, Greedy algorithms use locally available data to find an optimal solution. This algorithm is similar to the *Activity Selection* problem, but we must add two more dimensions to the problem. First, each rectangle has a left and a right edge, akin to a start and finish time. Second, each rectangle has a height, a bottom and a top edge, and thirdly, there can be any number of metal layers in the physical layout of a microchip. So in essence, this is a three dimensional problem, and the goal is to find all those rectangles that touch (connect) each other. The traversal algorithm is implemented as a method in the GST (Generalized Shape Traversal) engine class.

The local (greedy) data to abet decision-making consists of a list of dendrites that represent each metal layer we are interested in. The term dendrite is borrowed form the AI (Artificial Intelligence) literature. Each dendrite has a state, and it is either on or off to indicate whether the layer has any interesting data on the current scan line, also, each dendrite owns a list of unordered rectangles. The left and right ends of a rectangle represent the domain of the object, and its height is the range of the object. I use these mathematical terms throughout the algorithm because they convey the idea more clearly. As each domain (a rectangle from any metal layer) is encountered it is added to the dendrite corresponding to its layer in O (1). As the scan lines increase, the dendrites are kept trimmed by removing any domains whose ranges have ended. In this way, as each rectangle object is processed to check for connectivity, only the locally available data of the relevant dendrites is available.

The following is the UML (Unified Modeling Language) [3] Information Model diagram of the algorithm:

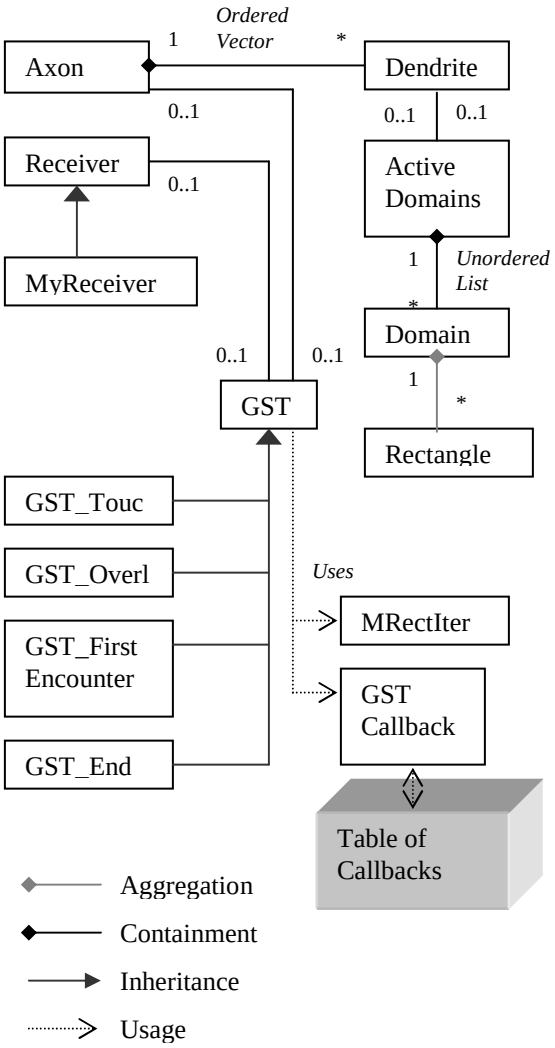

Figure 1.

The algorithm is semi-greedy because the potential for the local data to grow is high, especially for physical layers whose domains have large ranges. This is the case for vertically designed metal layers. To aid with this problem, as we traverse each relevant dendrite, I very quickly discard irrelevant data and only proceed to check for connectivity if the data cannot be discarded.

The algorithm preserves the *greedy choice property* because clearly the optimal solution is found by making local choices. The local choices use the information available on the trimmed 'on' dendrites of each layer. The algorithm also preserves the *optimal structure property* because as we sweep the artwork of the microchip, an optimal solution clearly exists up to that point. In other words, the algorithm could start and end at any scan line.

Description of classes

- GST: This is the class that maintains the main traversal engine and manages the callback mechanism as events occur.
- Domain: A Domain is a class that represents a true domain in the mathematical sense along the x-axis, where its definition requires a left and a right endpoint. The Domain object maintains a reference to the rectangle that it represents in the chip layout (see Figure 2), and a state variable that is used to help keep dendrites trimmed. As a scan line traversal proceeds, Domains are instantiated every time that a lower horizontal edge of a rectangle shape is encountered. When a scan line changes, the state of the Domain is changed to indicate that the rectangle object it references is still active, or if the scan line change causes the algorithm to encounter the top edge of a rectangle, then we know that we have reached the end of the rectangle and the Domain's state is updated to reflect this.
- Rectangle: a Domain aggregates a Rectangle class because the Rectangle objects are actually owned by metal layers of the chip not the Domain objects themselves. If a Domain object disappears it has no effect on the geometries of the metal layer.
- ActiveDomains: An unordered list of Domain objects.
- Dendrite: This structure manages the state of its *ActiveDomains* list. At any given moment during the GST traversal a Dendrite is either 'on' if it has features on the scan line being processed, or it is 'off'. If any given Dendrite is updated during a scan, then this indicates that at least one Domain in the ActiveDomains list has changed its state.
- Axon: A vector of Dendrites.
- MRectIter: This class is a multiple rectangle iterator. It manages a scan line traversal for every metal layer that the algorithm is processing. It returns the next left-most domain in the current scan line regardless of which metal layer it comes from.
- GST_Touch, GST_FirstEncounter, GST_End, GST_Overlap: These classes inherit all the GST properties, but most importantly they implement a pure virtual method to execute the appropriate callback when an event occurs. All these classes have an ISA relationship with GST because in this way they can be stored generically with the table of callbacks, thus taking full advantage of polymorphism.
- Receiver, MyReceiver: The Receiver class is abstract and defines pure virtual methods that must be implemented by a client application interested in processing the specific events.
- GST_Callback: Registers callbacks in a table.

Runtime diagram

The following diagram shows the run time relationships between some of the key data structures and the geometries that are referenced in the physical layout of a microchip. When a registered event is detected, the appropriate callback implemented by the *MyReceiver* class is called.

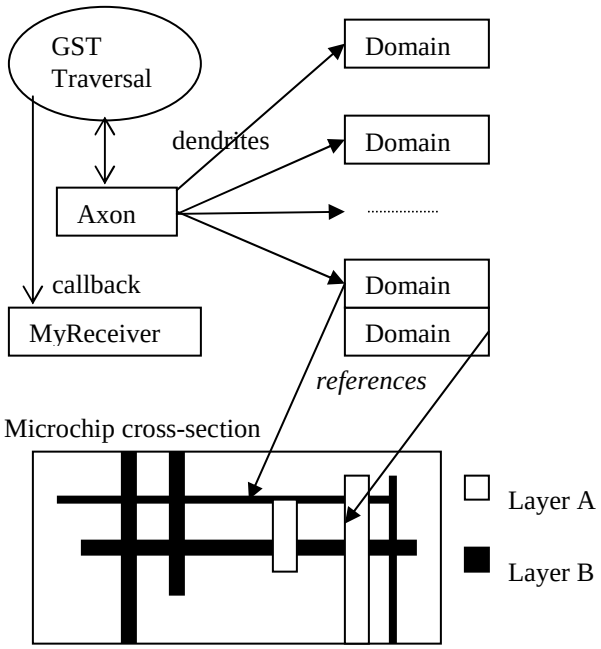

Figure 2.

Traversal Algorithm PseudoCode

The following is a description of the pseudo code of the GST traversal algorithm:

> Initialize all relevant structures
> Use MRectIter to open a scan based traversal of every registered metal layer.
> For every Domain object returned by MRectIter

do :

1) If the scan line changes, then update the state of every Domain object in any active Dendrite.
2) Add the new Domain object to its corresponding Dendrite in the Axon
3) Check for possible events such as touches, or overlaps and make callbacks appropriately
4) Post-process all dendrites. Remove from the dendrites any domains whose ranges have expired. This way we keep local data trimmed and greedy.

Callback Mechanism

Most existing algorithms use a traditional C solution of simple pointers to functions to build the callback mechanism for client applications. A client basically implements a function and passes its pointer to the scan-based engine. The engine, in turn, calls this function whenever it encounters an event of two shape objects. In this case, the shape objects happen to be rectangles and they represent wires or blocks in some layer of the chip. The client application then decides what to do with the event. Events include overlaps, touching, first encounters with shapes, etc.

This methodology is outdated and lacks proper extensibility characteristics. The motivation was to write an object-oriented mechanism that could be used by various CAD analysis tools and one that is flexible enough as to allow these tools to be modifiable by design. This following object oriented architecture provides a much richer API to clients of scan-based traversals.

Rather than registering a simple callback pointer, a client defines a class that implements the various callback methods. The client can then register with the server (scan-based engine) which of the events of the scan based traversal it is interested in. As the server performs its scan based traversals it can check which events the client is registered for, and make the appropriate call. This callback mechanism for microchip artwork was developed using the *command design pattern* [2], but has been adapted and modified to fit specifically with the scan based chip traversal algorithm. Each of the GST* classes inherits from the main GST class, but most importantly they implement the pure virtual *Execute()* API. It is this API that calls the correct method in the *MyReceiver* class. If a client application chooses to register for a particular event it must implement the method in the *MyReceiver* class, and register (via the *Register()* API) with the traversal engine. Upon registration, the traversal GST engine instantiates a GST* class and stores its pointer in a static callback table. The ISA relationship between these classes and the main GST engine class guarantees polymorphic behavior when retrieving the pointers to the callbacks.

Complexity Analysis

The previous section describes the pseudo code of the GST traversal algorithm. The algorithm has been annotated to facilitate its complexity breakdown. For the purpose of the analysis, the following key is used:

d = Dendrite (a list of Domain/Rectangle objects)
m = Metal layer
n = A rectangle object

Since Domain objects reference Rectangle objects, the analysis uses rectangles. A domain is merely a larger data structure that has no significance in terms of time complexity. Each of the four tasks annotated in the pseudo code are analyzed independently.

Task 1:

This task is executed only when a scan line changes. Empirical tests show that on the average there are O(n^2/3) scan lines per metal layer.

Since there is one dendrite per metal layer we have O(m) x O(d), but since the number of metal layers in a chip is small for the engines that use the GST traversal algorithm (~2 layers per run), then O(m) approximates a constant, so we can say that O(m) ~ O(1).

The number of rectangles per dendrite varies considerably depending on whether a metal layer is horizontally or vertically laid. Empirical tests have shown the number of rectangles per scan line to be O(n^1/3). So the complexity of task 1 is approximately = O(n^2/3) x O(n^1/3 /m)

Task 2:

This task is executed for every rectangle encountered by the GST traversal algorithm regardless of which metal layer it comes from. Finding the correct dendrite in which to add the rectangle is O(m), but since the number of metal layers in a chip is small and even smaller for the engines that use the GST traversal algorithm (~2 layers per run), then O(m) approximates a constant, so we can say that O(m) ~ O(1). Also, since we add the rectangle to the end of the dendrite list, this is a O(1) operation. So the complexity of task 2 is approximately = O(1)

Task 3:

We check for events every time we encounter a new rectangle by the GST traversal algorithm. The rectangle is checked against all dendrites, thus the complexity of this task is O(n) x O(d), and since the approximate number of rectangles per 'on' dendrite is in the order of [(n ^ 1/3) / m], then we have the complexity of task 3 be approximately = O(n) x O(n^1/3 / m)

Task 4:

This task has essentially the same time complexity as task 1. It only occurs when a scan line changes.

The total upper bound time complexity of the GST traversal algorithm is:

= task 1 + task 2 + task 3 + task 4
~= O(n) + O(1) + O(n^4/3) + O(n)
~=O(n^4/3)
~=O(n^1.33)

Note that experiments were carried out (see experimental section) where the data structure used to represent each dendrite was changed to accommodate binary trees, and sorted lists. No apparent advantages were found. Insertion into a binary tree or a sorted list was O(n log n) as opposed to O(1) (task2), however task 3 benefited in that we could check for events in an ordered container rather than for every rectangle contained in every dendrite. The latter yielded slightly better results for vertically laid layers where the average size of dendrites is larger, however, since traversal algorithms are used across many layers, some of which are horizontally laid; the ordered container did not yield any better performance.

Note that we have purposely left out analyzing the complexity of callbacks, as it is highly dependent on client applications.

Experimental Results

The following tables show elapsed times of running the GST traversal algorithm on real world chip data. We have chosen chips of various sizes with differing metal layer sizes. Two values are displayed for each run, one is the time spent by the system running the command, and the other is the time spent executing the command. Note that times can vary depending on the performance of the memory in which the program is running, however the relationship between those times stay similar.

Table 1: Results for finding touching and overlapping rectangles in a single layer.
M: mixed layer
H: horizontal layer
V: vertical layer
u: user time
s: sys time

No. of Rectangles	*GST Traversal Algorithm*	*Ordered Container Algorithm*
11446 **H** 598 Touches 19216 Overlaps	5.62 *secs.* (**u**) 2.70 *secs.* (**s**)	5.79 *secs.* (**u**) 2.83 *secs.* (**s**)
20660 **V** 0 Touches 29665 Overlaps	6.11 *secs.* (**u**) 2.91 *secs.* (**s**)	6.17 *secs.* (**u**) 2.98 *secs.* (**s**)
888137 **M** 517050 Touches 0 Overlaps	37.55 *secs.* (**u**) 10.25 *secs.* (**s**)	23.00 *secs.* (**u**) 10.49 *secs.* (**s**)

Table 1 shows results within a single layer. We compare results between the GST traversal algorithm described in this paper against an algorithm that uses ordered containers to store rectangle information.

The ordered container algorithm actually uses a sorted list. The list is sorted by the leftmost x-coordinate of the rectangle. The GST algorithm is faster inserting rectangles but is slower checking for events (touches, overlaps). What we can extrapolate from this table is confirmation that for vertically laid layers the GST semi-greedy algorithm does not perform as well as for horizontally laid layers. Vertically laid layers contain large numbers of *stretched out* rectangles in the y-axis direction, thus forcing the algorithm to keep this information around longer and making the search for events more time consuming. The opposite is true for horizontal layers, where the rectangles are *stretched out* along the x-axis, and the data maintained is always small, thus the better performance. As the number of rectangles increased to ~ 1M, we see an increase in user time, which can be attributed to the callback mechanism.

Table 2: Results for finding touching and overlapping rectangles across two layers.
O: Layers are orthogonal
u:user time
s:sys time

No. of Rectangles	*GST Traversal Algorithm*	*Ordered Container Algorithm*
32106 **O** 946 Touches 159987 Overlaps	7.30 *secs.* (**u**) 3.51 *secs.* (**s**)	7.16 *secs.* (**u**) 3.48 *secs.* (**s**)
98445 598 Touches 221399 Overlaps	8.38 *secs.* (**u**) 5.05 *secs.* (**s**)	8.42 *secs.* (**u**) 5.35 *secs.* (**s**)
107123 0 Touches 1 Overlap	6.70 *secs.* (**u**) 2.40 *secs.* (**s**)	6.15 *secs.* (**u**) 2.56 *secs.* (**s**)

Table 2 shows results when finding touching and overlapping events across two layers. This experimental case is the most typical case used by CAD engineers when designing analysis tools. Here we notice that both algorithms have similar performance. This is due in part to the fact that adjacent metal layers in a microchip tend to be laid out orthogonal to each other in order to maximize routing. Test cases where two layers are not necessarily orthogonal to each other are not labeled *'O'*. This is the case for example when an application is trying to find overlaps between a metal or poly-silicon layer with a *contact* layer. A contact layer is a layer that maintains the vias used to route and connect two or more adjacent layers.

Even though we only show experimental results for two layers, there are cases where some analysis tools gather information from several layers at one time, but this is not the norm. It is important to note that you can

find various cases where these traversal algorithms are not necessarily optimal for the tasks at hand and solutions that require different approaches are necessary beyond just different containers. Examples of other algorithms used in this space include *quad trees, nearest neighbor, corner stitching*, etc. Excellent sources of information are [4] and [5].

Advantages

The following are some of the key advantages implemented by this algorithm.

- Keeps track of Domain objects on a per layer basis rather than scan line. This way we could query a layer at any time for its current state.
- Allows for reflexive traversals by specifying a single layer.
- Can specify multiple layer associations and the algorithm will report the connectivity of each of these associations with a single pass through the artwork of the microchip.
- Slightly better performance for horizontally layed artwork. Greedy information is smaller.
- The Dendrite/Axon concepts borrowed from the Artificial Intelligence field lend themselves to further research. Lots of additional information could be tracked as we traverse data.
- Parameter passing of the objects causing the event. This allows a client application to receive rich data, which it can further process if it chooses to.
- Full encapsulation of the callback mechanism.
- Easily extensible from a server perspective. You basically just create a new class to handle the new events. The GST Engine takes full advantage of polymorphism.
- Easily extensible from a client perspective. You just add a new API to handle the new callback without having to change the class schema.
- No need to re-compile the client applications if the server is upgraded to report new events.
- Decouples the callback objects from the engine performing the layout traversals.
- Because each callback is an instantiation of an object, you can store this information (i.e. serialize it) for latter use. For example, this information could be used to undo complex operations.
- The client application can define its own substructure to handle events.

Conclusion

There exist numerous analysis tools that are based on shape traversal algorithms and reporting mechanisms.

It is the combination of efficient algorithms with extensible callback architectures that provides the biggest benefit for CAD analysis tool developers. Also, it is the improvements performed on the core architectures and engines of CAD that leads to improved and shorter software lifecycles of analysis applications. Although any number of variations exists to improve performance or containment, no single solution can anticipate the requirements of all analysis applications, thus optimization techniques are necessary to provide good enough solutions that allow for generic usage and allow for development of tools in a timely manner. Examples of analysis tools that make use of scan-based algorithms include DRC (Design Rule Checking), ERC (Electrical connectivity and Rule Checking), and parasitic extraction.

Future work of this engine/architecture involves its usage in CAD analysis tools such as multi-layer capacitance extraction to calculate overlap, fringing, and shielding interactions between multiple layers, and tools to perform multiple logical operations (And, Or, AndNot, etc.) as one command to eliminate multiple passes through shape data.

References

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, Introduction to Algorithms, McGraw Hill, 1990.
[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley, 1995.
[3] Booch, Jacobson, Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, 1999.
[4] Neveed Sherwani, Algorithms for VLSI Physical Design Automation, Kluwer Academic Publishers, 1997.
[5] Sabih Gerez, Algorithms for VLSI Design Automation, Wiley, 1999.