Improving Display Speed in Electric

Steven M. Rubin
Sun Microsystems Laboratories

October 17, 2005
Memo SML#: 2005-0523

Abstract

There are many computationally intensive tasks in the design of integrated circuits. One of the simplest yet most important is circuit display. Display speed is critically important to a design system because users do it frequently and expect rapid response. A typical requirement is the ability to view an entire chip, consisting of millions of transistors. This can tax the limits of any software or hardware system.

The Electric™ VLSI Design System is an integrated circuit design program that has been in use for over 20 years. During its lifetime, Electric has had three different display algorithms, each faster than the other. Only after the third version of circuit display was coded could Electric compete with commercial tools in terms of display speed. This paper describes all three display algorithms and gives performance figures for them.

Introduction

The Electric™ VLSI Design System has existed since 1983 [Rubin83]. When first written, it was coded in the C programming language. Even in 1983, however, portability was considered important. To achieve this, an abstract API was used to describe the operating system interface. For example, rather than issuing the operating-system-specific call to draw a line, Electric called its own "drawline" routine. Each version of the operating-system interface had a different implementation of "drawline" which then made the proper operating-system-specific call. By 2003, there were five complete implementations of the operating-system interface, which could operate on Windows, Macintosh, and three variants of UNIX/Linux (Athena, Motif, and Qt).

The use of a common interface to the display had the drawback that most display optimizations could be implemented only if they were coded 5 times, once for each of the implementations of the OS interface. Therefore, for the first 20 years of its life, Electric had a single display algorithm, which did no optimization. This "Brute Force" display system examined every level of the circuit hierarchy in order, drawing everything that it found. It could take minutes to display a large chip.

In 2003, Electric was recoded from C to Java [Rubin04]. This had a number of advantages, not the least of which was the fact that there was now a single operating-system interface: Java Swing. Unfortunately, initial implementations of the display algorithm had even worse performance than the equivalent C code. Thus began the quest for a more intelligent display algorithm.

Electric's Display

Before describing the improvements to the display algorithm, it is useful to describe the functionality demanded by the design system.

Integrated circuits make use of hierarchy to save memory and to organize them conceptually. Each level of the hierarchy is called a "cell," and contains a relatively small amount of circuitry. By placing instances of cells into other cells, hierarchy is established, and complex circuits are described. It is often the case that a single cell is instantiated hundreds of times in the integrated circuit chip, so a full display of the chip will examine that cell repeatedly.

To make matters more complex, there are many different "layers" in an integrated circuit, and designers want to be able to see them as they overlap and interact. Most commercial systems draw each layer as a patterned area, choosing the pattern so that it will not interfere with other patterns from other layers. Because the areas are patterned, it is common to draw a solid border around the pattern so that its edge can be clearly distinguished. The resulting display is often very confusing, and users frequently remove layers from the screen in order to be able to focus on those that have the most interest.

Electric is able to draw patterned layers, but it is also able to draw them in solid colors. Where two solid colors overlap, a third color is drawn, which shows the overlap. Layers that can interact with others in this fashion are called "transparent". Determination of the overlap area can be computationally intensive, so Electric solves the problem by using a color map. Each transparent layer is rendered into its own bit plane of the display. The combination of bits from all of the transparent planes forms an index into a color map, and the entry in the map determines the color to display. By precomputing the overlap colors, combinations of layers appear to be transparently mixed.

Unfortunately, it is not possible to draw all layers transparently. One reason for this is the large number of layers that exist in most chips. If, for example, there are 32 layers being drawn, then the color map must be 2^{32} entries long, which is excessive. Another reason for not using transparency everywhere is that there are not enough different colors for them to all overlap and still be recognizable by the user. Eventually, it all will look "brown". Thus, Electric uses transparency for the important layers (usually just the 5 most important ones) and uses opaque patterns for the other, less important layers.

As an example of this "Brute Force" display, Figure 1 shows a chip that has been expanded all the way to the bottom.

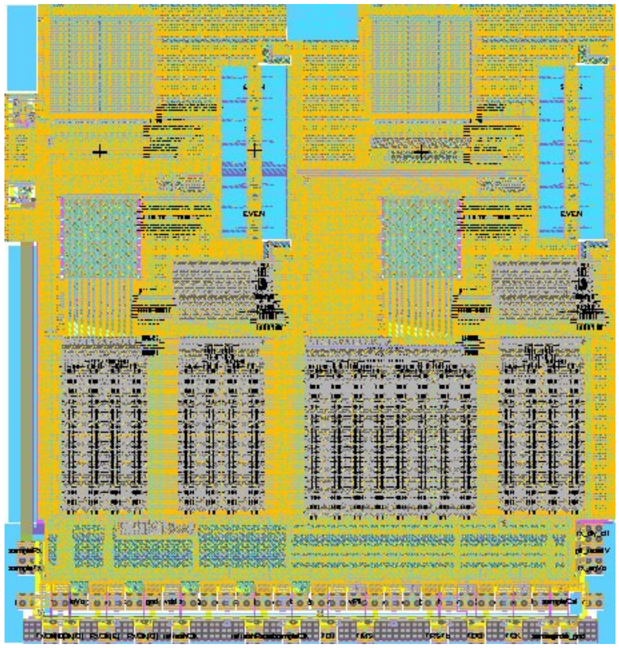

Figure 1: Brute Force Display
This is the qFourP2 chip, which has 1 million transistors. This image took over a minute to produce.

Basic Optimizations

One obvious optimization, that was made to all of the display algorithms, is to use the cell hierarchy to clip parts of the chip that are offscreen. As the hierarchy is explored, any subcell that is completely out of the display does not need to be examined at all. This optimization saves great amounts of time when the display is zoomed into a small part of the chip, but has no effect on whole-chip display (see the “Brute Force” timings in Table 1).

After building the Java implementation of Electric, it was discovered that the display could be faster if custom drawing code was written. Rather than use the Java Swing methods for drawing layers, an offscreen buffer was constructed, and custom code was written to fill that buffer. Because Electric uses a transparency model for display, the Swing rendering was very expensive. Using an offscreen buffer improved display speed by a factor of three.

Another optimization was to simplify (or “greek”) very tiny objects. Some components have many complex layers associated with them. But when such a component is only 1 or 2 pixels in size on the screen, there is no point in trying to show that complexity. In such a situation, the component is approximated with a single dot, drawn in an appropriate color. The same thing is done when a wire is long but very narrow. This simplification improved display speed by a factor of two.

Beyond greeking, another optimization has been added. If the system is in the middle of displaying, and a new display request is made, the current one aborts. This happens in smooth panning or scaling, where mouse drags alter the display. If the mouse changes happen faster than the display can keep up, then each redisplay is aborted. Visually, only parts of the display are shown, but they appear to move continuously. When the mouse pauses, the full display is shown.

Caching

No optimization can solve the basic problem of having too much to display. Something else is needed to get the level of performance demanded by users. Common information must be saved so that display computation can be reduced. The question is: what should be cached? Two different caching algorithms were coded.

The first caching algorithm saves the contents of a cell, so that its display can be more quickly drawn after the first time. This caching algorithm, called "Pixel Caching" saves the actual bits on the screen. When a cell is first drawn, its bits are saved so that they can be instantly copied to subsequent locations. This has the advantage of very quick display, but is sensitive to changes in scale, layer visibility, and other display controls.

The second caching algorithm, called "Vector Caching" caches the transformed geometry, prior to being rendered on the screen. Although somewhat slower in some situations, it is surprisingly fast in most situations, and does not degrade when the scale changes.

Pixel Caching

The first attempt at display speedup caches the image of a cell when it is rendered (see Figure 2). Because of the transparency model, it is not possible to simply copy the bits in a cell. This is because other transparent layers might be drawn over the cell, and need to combine with the transparent layers inside of the cell. Therefore, the offscreen cache of a cell's image consists of both transparent data and opaque data. The final compositing of this information is done only when the full display is ready to be shown.

One problem with Pixel Caching is that cells may be drawn in different orientations. If one instance of a cell is rotated and another is not, then they have very different appearances on the display. To solve this, cells are cached many times, once for each different orientation on the display. This might seem like an excessive number of caches, but in most integrated circuit designs, cells appear in only the "Manhattan" orientations (multiples of 90 degrees) and therefore the maximum number of cell caches is 8 (including mirroring).

Another problem with Pixel Caching is that the precise location of the cell on the display may be wrong. For example, when rendered in a particular location, a cell may be 20 pixels wide, but when rendered elsewhere, it may be 19 or 21. This is caused by differences in subpixel alignment. However, because these errors are never more than 1 pixel in size, they are not considered important.

Yet another problem with Pixel Caching is that patterned layers will not tile cleanly when the patterns are copied elsewhere on the display. Large background layers such as well or substrate are often drawn as patterns, and are drawn in every cell. When these cells are copied to different locations on the screen, the patterns will not line-up at cell borders, leaving a visually disturbing artifact. The solution is to save patterned layers as bitmaps (similarly to the way that transparent layers are recorded) and to convert these layers to patterns at the end of the rendering process.

An important consideration in Pixel Caching is the memory that will be consumed by these offscreen images. Two optimizations are implemented to save memory. First, if a cell appears only once, then its image is not cached. In order to detect such situations, the display algorithm must scan the entire hierarchy and count the usages of each cell. If the display is redrawn at the same scale, however, then the cell does get cached because even though it appears only once in the circuit, it is being drawn multiple times and can benefit from being cached.

The second memory-saving optimization is that a cell is only cached if its on-screen size is not excessively large. The rule currently is that any cell whose width is greater than half of the display width AND whose height is greater than half of the display height, is too large to cache.

The most unfortunate aspect of cell image caching is that the saved cell images must be recomputed frequently. If the scale of the display changes because of a zoom in or out, then all of the images must be abandoned. If the user changes the visibility of a given layer, then the images must also be recomputed with the new visibility. Even minor display artifacts, such as text settings, can cause the images to be abandoned. Also, if two different windows are viewing the same circuitry, and are at different scales, then the system must constantly reconstruct the cache (or it must maintain separate caches for each window). Recomputing these images is time and memory intensive, and can add significant delays to the display time.

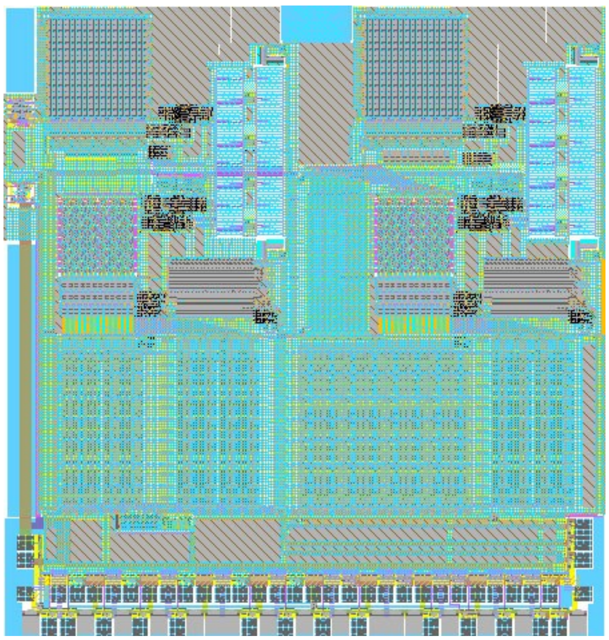

Figure 2: Pixel Caching Display

This is the qFourP2 chip, rendered with the Pixel Caching algorithm.

Vector Caching

It was eventually decided that Pixel Caching was the wrong way to do rapid display. The algorithm has excessive complexity, and is overly sensitive to changes in scale (a frequently used operation). For example, when users run long wires in a circuit, they zoom into an area, select the end of a wire, zoom out, pan over to a new location, and then zoom in to work on the other end of the wire. Thus, scale changes occur nearly as often as position changes.

What is needed is a cache of a cell that is scale-independent. As it turns out, rendering the contents of a cell involves a large amount of computation and memory allocation, because it is nontrivial to convert the cell contents to display coordinates. By caching these coordinates, just before they are scaled for the display, a scale-independent cache of the cell is obtained. This is the Vector Cache.

As with Pixel Caching, each orientation of a cell is saved separately. This avoids expensive transformations (both constructing them and applying them). References to subcell instances simply give an offset of that instance, and the orientation of that instance relative to the current cell. This orientation is combined with the orientation of the current cell to select an absolute instance orientation, which is rendered.

Although construction of the Vector Cache takes time, the cache is valid for as long as the cell remains unchanged. The only thing that invalidates the cache is when the user changes the contents of the cell.

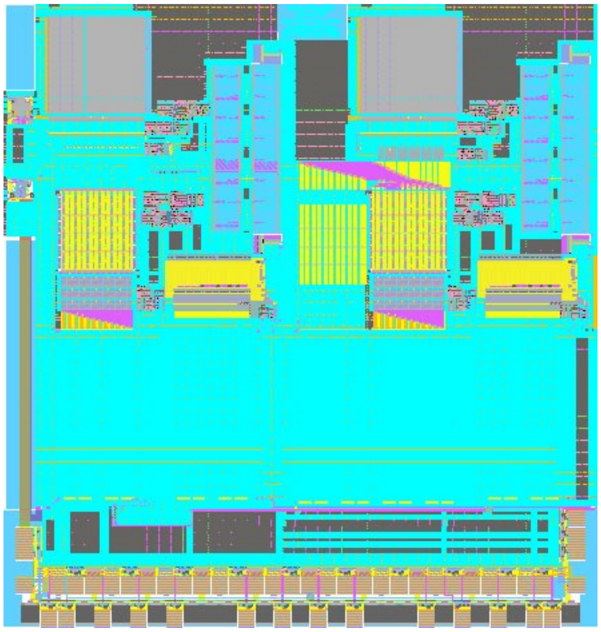

Figure 3: Vector Caching Display
This is the qFourP2 chip, rendered with the Vector Caching algorithm.

Cell Greeking

A common abstraction in full-chip display is the “greeking” of cells when their contents are too small to see. The Brute Force display algorithm greeks cells when their scale is greater than 4 grid units per display pixel. This means that, regardless of circuit complexity, when 4 units of the design space are smaller than a pixel, all cells are drawn as gray patterned boxes. This gives very rapid display when zoomed-out, but little information is conveyed by the display (see Figure 4).

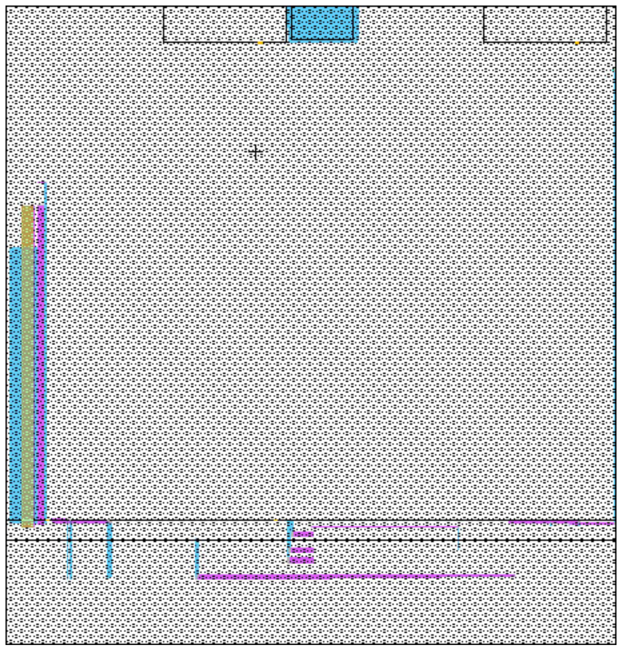

Figure 4: Brute Force display, with “greeking”

This is the qFourP2 chip, displayed by the Brute Force algorithm. At this scale, every cell is “greeked” with a gray stipple pattern. The only visible parts of the chip are the top-level wiring.

The Pixel Caching algorithm does not need to do cell greeking because every cell is already captured as an image. The Vector Caching algorithm does have to do cell greeking, because without it, all circuitry will be rendered, regardless of its size.

Because of the inadequacy of the Brute Force algorithm’s cell greeking method, it was decided to look for better greeking techniques for the Vector Caching algorithm. The first improvement was an adaptive determination of when a cell should be greeked. Rather than greeking at a fixed scale, each cell is greeked at a different scale, according to the cell’s contents. If a cell contains large objects, such a pads, then even when zoomed far out, that cell will have its contents drawn. During construction of the Vector Cache, the maximum feature size of a cell (and all subcells within it) is saved. At display time, if every object inside of a cell is too small, then the cell is greeked. Because implant layers are usually larger than other objects in a cell, these layers have their “feature size” reduced so that cell greeking occurs correctly.

There are two exceptions to the new cell-greeking rule:

1. If a cell is larger than 10% of the area of the display, it is not greeked.
2. If a cell is an icon (found in schematics), it is not greeked.

In an attempt to improve on the gray pattern used to greek cells in the Brute Force algorithm, three different cell greeking methods were explored:

- Choosing a better greeking color. Instead of gray, the greeking color for a cell is determined by examining the contents of the cell (all the way to the bottom of the hierarchy). The area of every layer is summed up, and the greeking color is the weighted average of all of the layer colors. This generally chooses a bland color that mixes all layers, but does not capture the appearance of any of them. Also, the variations in cell colors are not pronounced-enough to visually distinguish different cells. Therefore, the resulting display is an unpleasing collection of colors that vary only slightly.
- Use proper layer colors in a random pattern. Once again, the area of every layer is summed-up. The greeked cell is then drawn by randomly selecting layers inside of that cell. The frequency of a layer's color is weighted by the relative area of that layer inside of the cell. The colors look correct, but the random patterns give the circuit a noisy appearance, which is distracting.
- Build a small "greeked" image of the cell. The greeked image is 25x25 pixels in size, and it contains a rendering of the cell. This greeked image can be quickly scaled-down to fit over the cell being rendered. It can be used in any situation where the cell is 25x25 or smaller, so for cells with very little circuitry, these greeked images will save display time. This produces the most pleasing images (see Figure 5). The only downside of these images is that they have to be recomputed when layer visibility changes.

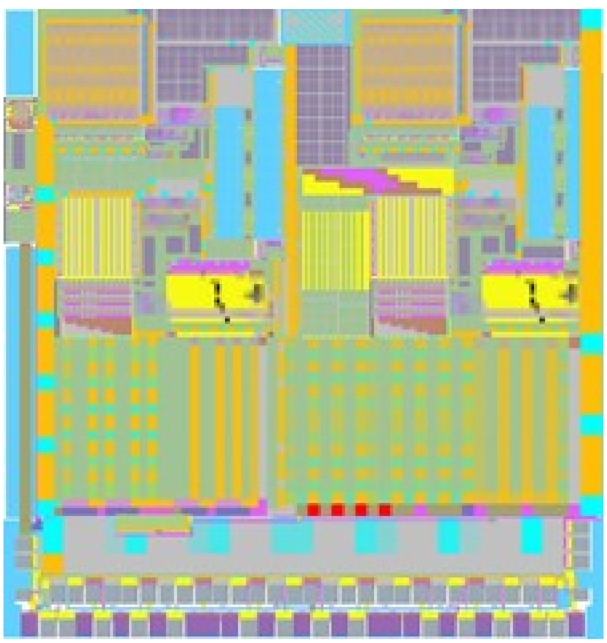

Figure 5: Vector Cache Display, with "greeking"

This is the qFourP2 chip, displayed by the vector cache algorithm. Cells are "greeked" with colors that have been automatically chosen to represent them. Different cells are distinguishable, but no individual components are visible.

Results

To test these display algorithms, four different chips were rendered. The chips ranged in complexity from 142,000 transistors up to nearly 1-million transistors. These chip were expanded fully and drawn at different magnifications.

Table 1 shows the times for rendering these chips at different magnifications and with different display algorithms. The Vector Caching algorithm performs best in nearly all situations. Here are specific notes on the way the three display algorithms were tested:

- For the Brute Force algorithm, cell greeking was disabled (or else scales "Full" to "In 8" would have greeked everything, taking no time, and showing no information).
- For the Pixel Caching algorithm, two columns are shown. The first column is the time taken to display and rebuild the cache (this happens on the first display after changing the zoom scale). The second column is the time taken to display once a zoom scale has been cached.
- For the Vector Caching algorithm, there is a one-time delay that happens when the chip is first expanded (shown in the column title under the word "Vector"). After that, there is no difference in display speed between zooming and panning.

Another performance consideration is the amount of memory used. Table 2 shows the memory consumption for the two caching algorithms on the different chips. The Vector Caching algorithm takes more memory, but the Pixel Caching algorithm consumes and destroys smaller amounts much more often. In a single set of zooms from full display to the highest magnification, the Pixel Caching algorithm runs through much more memory than the Vector Caching algorithm.

The final experiment that was run was a test to find the optimum cell greeking image size. Table 3 shows the rendering times for the Sport chip. It can be seen that the 25x25 pixel image uses a small amount of time for initial cache construction, but still provides good performance when displaying.

Conclusion

Three display algorithms have been coded in Electric. The Brute Force algorithm cannot compete with the other two, which cache information for rapid display. Of the two caching algorithms, the Vector Caching is superior in overall speed and memory usage. In addition, users are much happier with the performance of this algorithm. It is the standard display algorithm for Electric, version 8.03 and higher.

	qFourP2 (957,324 transistors)				Loco (647,582 transistors)			
	Brute Force	Pixel		Vector (21.704)	Brute Force	Pixel		Vector (5.469)
Full	74.6	1.078	0.187	0.671	39.468	0.328	0.141	0.156
In 2	33.3	1.422	0.328	0.609	20.672	0.250	0.156	0.297
In 4	11.2	1.563	0.625	0.672	9.657	0.578	0.391	0.344
In 8	2.17	1.297	0.562	0.469	3.015	1.062	0.438	0.781
In 16	0.406	0.844	0.344	0.140	0.984	0.329	0.281	0.344
In 32	0.079	4.516	0.141	0.047	0.375	0.312	0.297	0.172
In 64	0.047	0.547	0.156	0.031	0.203	0.360	0.297	0.125
In 128	0.031	0.125	0.156	0.031	0.172	0.390	0.312	0.125

	qLite (546,535 transistors)				Sport (142,648 transistors)			
	Brute Force	Pixel		Vector (12.953)	Brute Force	Pixel		Vector (4.325)
Full	37.609	0.735	0.343	0.406	3.859	0.531	0.188	0.297
In 2	28.281	0.859	0.532	0.625	1.719	0.515	0.344	0.531
In 4	11.046	0.734	0.344	0.468	0.422	0.578	0.312	0.422
In 8	3.516	0.656	0.297	0.828	0.125	0.406	0.250	0.141
In 16	0.937	0.406	0.328	0.375	0.062	0.500	0.203	0.047
In 32	0.218	0.422	0.250	0.109	0.093	0.234	0.203	0.047
In 64	0.125	0.328	0.187	0.031	0.078	0.093	0.093	0.047
In 128	0.078	0.141	0.140	0.062	0.078	0.094	0.078	0.046

Table 1: Rendering times for different algorithms
Four different chips were displayed (qFourP2, Loco, qLite, and Sport) using three rendering algorithms (Brute Force, Pixel Cache, and Vector Cache) and 8 different magnifications (full chip to zoomed-in by a factor of 128). Times shown are in seconds, when run on a Sun 2100Z workstation, with 6 gigabytes of memory and dual Opteron 246 processors.

	qFourP2	Loco	qLite	Sport
Vector	85.1	22.8	52.7	19.6
Pixel Full	32.4	13.8	25.2	5.8
Pixel In 2	26.0	9.7	7.9	3.0
Pixel In 4	37.7	22.8	28.0	13.0
Pixel In 8	19.5	35.2	7.3	8.8
Pixel In 16	35.7	5.9	18.4	11.0
Pixel In 32	64.2	6.1	18.5	3.1
Pixel In 64	18.3	9.0	19.4	3.3
Pixel In 128	3.3	8.9	16.1	2.6

Table 2: Memory consumption for two caching algorithms
Because Vector caching is done only once, there is only one figure for the size of its cache. Pixel Caching is done each time scale changes, so the size of each magnification's cache is shown. Cache sizes are in megabytes.

Size	Initial	Full	In 2	In 4
10x10	3.895	0.640	1.172	0.422
15x15	3.922	0.281	0.984	0.437
20x20	4.157	0.282	0.953	0.422
25x25	4.406	0.312	0.562	0.437
30x30	4.688	0.328	0.609	0.422
35x35	5.219	0.343	0.625	0.469
40x40	5.750	0.281	0.656	0.500
50x50	6.672	0.297	0.734	0.500
60x60	8.219	0.328	0.843	0.547

Table 3: Vector Caching performance for different Greeked Cell sizes
The size of the "greeked cell" is varied from 10x10 pixels up to 60x60 pixels. The times shown are in seconds. Timings are shown for the "initial" cache construction, and display of the full chip, zoomed in by 2, and zoomed in by 4. Scales larger than this are not affected by the cell greeking.

References

- Rubin, Steven M., "An Integrated Aid for Top-Down Electrical Design", Proceedings, VLSI '83 (Anceau and Aas, eds.), North Holland, Amsterdam, 1983.
- Rubin, Steven M., "The Journey from C to Java: Experiences Translating Electric", Sun Microsystems Laboratories report SML 2004-0555, September 2004.