CADAPPLETS – Visualization of VLSI CAD Algorithms

John A. Nestor

***Abstract*—This paper describes the development of visualization aids for VLSI Computer-Aided Design algorithms. Each visualization is implemented as a Java applet that illustrates the problem formulation used for a given CAD tool as well as the details of how the algorithm executes. The general approach follows that of software algorithm animation, but greater emphasis is placed on the relationship of the algorithms to the problem formulation and underlying physical representation. The applets have been used in VLSI Design course lectures and are available on the web as a reference resource.**

***Index Terms*—Visualization, Animation, VLSI CAD, Design Automation, Placement, Routing.**

I. INTRODUCTION

THE ongoing revolution in VLSI technology has made possible the design of increasingly large and complex integrated circuits. Current chip designs can contain over 100 million transistors, and larger chips are expected in the future. The computer and communications products that result from these advances have had an impact throughout society.

Chip designs of this complexity would be impossible without the extensive use of Computer-Aided Design (CAD) tools to automate key parts of the design process. Fig. 1 shows a simplified diagram of CAD tools used in the design of *semi-custom* VLSI chips – a popular style in which a pre-designed library of *standard cells* is used.

Input is in the form of *a Hardware Description Language (HDL)*, which specifies desired function. *Synthesis* tools translate this input into hardware, optimize the hardware to meet area and timing constraints, and output a *netlist* that specifies the hardware as a set of standard cells and connections (nets). Next, *Physical Design* tools assemble the cells and connections into a *layout* that is suitable for fabrication. Physical Design is usually performed in two steps: *placement*, which assigns specific positions to each cell, and *routing*, which defines the physical connections between the terminals of the cells using a limited number of metal layers.

The development of these CAD tools has been the subject of intensive research [1], [2]. The general approach to is to first develop a *problem formulation* that models the design problem and provides metrics for estimating the quality of solutions in terms of area, timing, power, and other concerns.

J. A. Nestor is with the Electrical and Computer Engineering Department, Lafayette College, Easton PA 08807 USA.

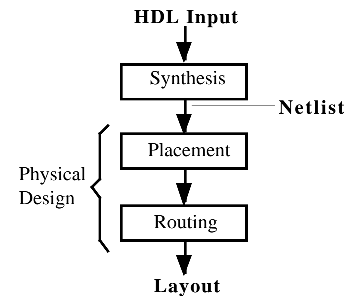


Fig. 1. Simplified Design Flow for Semi-Custom Chips.

Next, an algorithm is selected and applied to the problem formulation. Because of the size and difficulty of the problems attacked by these tools, tradeoffs are necessary. Problem formulations are often simplified in order to make them more tractable at the expense of design quality. Even with simplified formulations, exact algorithms are often impractical, so heuristics and probabilistic algorithms are common. Thus the quality of a design produced by a first pass through a set of CAD tools may require improvement before it is considered acceptable and the design process is usually iterative.

In order to use VLSI CAD tools effectively, designers must understand the tools' capabilities and, more importantly, their limitations. This in turn requires an understanding of how the tools work internally and arrive at a given solution.

The goal of the CADAPPLETS project [3], [4] at Lafayette College is to adapt ideas from software visualization and algorithm animation [5], [6] to aid students, designers, and CAD tool developers in gaining such an understanding.

While previous approaches to software visualization and algorithm animation provide a good starting point for this project, there are several differences between the study of "pure software" algorithms and CAD algorithms that must be considered when creating animations of CAD algorithms:

- There is usually an underlying physical representation of the work performed by each CAD tool that should be reflected in the visualizations.
- Problem formulations for individual tools often grossly simplify the physical representation to make a design task more manageable at a cost in degraded solution quality. Good visualizations should communicate both the form of the simplification and the quality loss.
- Even with simplified formulations, many design tasks remain intractable, making heuristics and probabilistic algorithms (e.g., simulated annealing) popular. Visualizations are needed to support these approaches.

This paper describes the general approach used by the CADAPPLETS project and surveys the visualizations that have been developed to date. Each visualization is implemented as a Java applet and is intended to provide users with an understanding of the general problem formulation used by a CAD tool as well the operation of one or more algorithms which use this problem formulation. The current focus of the project is on physical design algorithms for placement and routing, although synthesis-related demonstrations are planned for the future.

This paper is organized as follows. Section 2 reviews software visualization and previous efforts to animate CAD algorithms. Section 3 reviews the placement problem in VLSI CAD and describes a visualization of placement of modules with varying shapes using two algorithms: iterative improvement and simulated annealing. Section 4 reviews the routing problem and presents visualizations of two classic approaches to routing with very different formulations: *maze routing*, and *channel routing*. Section 5 discusses our experience using the visualizations in teaching and its use by others over the Internet. Section 6 concludes the paper and provides suggestions for future work.

II. Background

Algorithm animation and software visualization have been the topic of extensive research. In the mid-1980's Brown and Sedgewick developed the BALSA system [5] as a framework for software algorithm animation. More recently, many other systems have been developed; a good overview is provided in [6]. BALSA and similar systems animate software algorithms by presenting *views* - graphical renderings of data structures and execution history. As an animation proceeds, the position, shape, and color of objects in the view are changed to illustrate data structure changes, plot historical information about execution, and present statistics about the results of the algorithm. Multiple views allow the emphasis of different features of the algorithm in the different views.

The graphical views are tied to an algorithm to be visualized using the *interesting events paradigm*, in which algorithm code is annotated to identify important steps in the algorithm. As these events are reached during execution, the graphical views are updated accordingly.

With the advent of the world-wide web, it has become possible to provide visualization tools over the Internet, most commonly using Java [7] applets. These can be implemented as part of a large animation framework (e.g., [8]), or can be written as free-standing applets using the graphical user interface capabilities.

While these techniques have been used extensively for software algorithm animation, only a few animations of VLSI CAD algorithms have been developed. The N_ABLE project [9] and JADE [10] system provide applets that illustrate concepts in switching and automata theory and logic synthesis. The Blue Macaw tool provides visualization of fixed-size cell placement using simulated annealing [11]. A few other physical design animations have been developed as part of student projects in VLSI CAD courses [12, 13].

Limited visualization aids are also built into commercial CAD tools. For example, placement tools often use a *rat's nest* diagram [2], in which nets are displayed as straight-line segments laid over a diagram of cell placements. This gives the user a way to grasp the size of the nets and the relative congestion of different parts of the placement. Routing tools often display the end result of the routing, or provide displays that can be used to select wires for rerouting. However, these displays are intended to provide the user with feedback about the end result and not the operation of the algorithm.

In contrast, the visualizations developed in the CADAPPLETS project are intended to provide insight about both the operation of the algorithms as well as the end result. Displays like the rat's nest are used in these visualizations, but are augmented to illustrate dynamic behavior.

III. Visualizing Placement

The goal of placement is to assign physical locations and orientations to a set of connected cells while minimizing a cost function that measures chip area and routing quality. Depending on the design style used, cells may be of varying size, or may be fixed in size in one or two dimensions. Cells may be rotated or reflected about an axis.

The typical problem formulation for placement represents cells as rectangles on a planar surface with a cost function that provides a measure of placement quality - usually some combination of chip area and routing quality. Chip area is easily measured by finding the bounding box containing the cells. Routing quality is more difficult to measure. The ultimate measure is the length of the routed nets, but it is not practical to run a routing tool each time a placement is changed. Instead, *estimates* of routing quality are used that predict wire length and congestion.

Common net length estimates include the length of a minimum spanning tree or Steiner tree that connects the terminals of a net, and one-half the perimeter of a bounding box that contains the terminals of a net. A common congestion measurement is the count of the number of nets that cross a set of cut lines on the layout surface. Cost functions often also include a measure of cell overlap as a *penalty function*. These measures are combined in a weighted sum to form a complete cost function.

Given a formulation, the goal of a placement algorithm is to find an assignment of locations to cells such that the cost function is minimized. Several techniques have been proposed, including *iterative improvement* techniques which find a low-cost placement by making a sequence of small changes, *partitioning-based* techniques, which recursively break a design into minimally-connected groups of cells, and *analytical* techniques which minimize a cost function directly by solving a system of equations.

The first step in creating a visualization tool for placement is to provide a graphical representation that depicts the problem formulation. Fig. 2 shows the placement view developed for the CADAPPLETS project. It displays a "rat's

nest diagram" which displays cells, cell terminals (small black rectangles), and nets plotted as straight-line connections between the terminals. A status bar at the top of the display shows area, overlap, estimated wire length (currently using the half-perimeter estimate), and overall cost, which is a weighted sum of these values. The placement display can be manipulated by the user to explore the impact of changes - each cell can be rotated or mirrored by clicking the mouse over the cell, and each cell can moved to a different position by clicking and dragging.

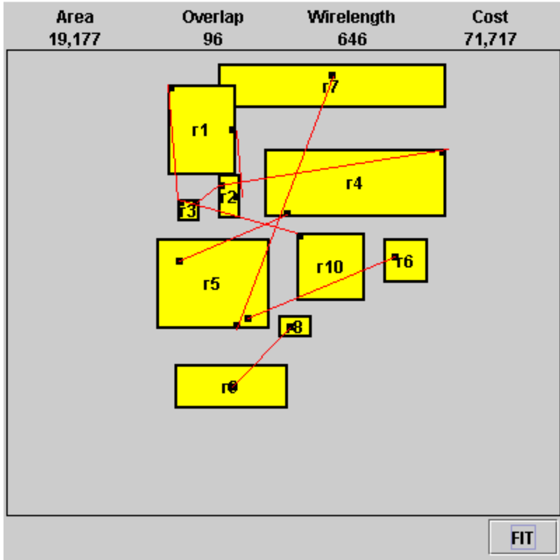


Fig. 2. Placement Display. The body of the display is the classic "rat's nest" diagram. Cost metrics are shown at the top of the display.

A. Placement by Iterative Improvement

Iterative improvement algorithms search for a good placement configuration by applying a set of simple *moves* – transformations that alter the placement in some way. While the details of iterative improvement algorithms vary, the key idea is to repeatedly apply randomly selected moves, while evaluating the impact of each move of the cost function. Table I shows a typical set of moves.

"Downhill" moves that decrease the cost function are *accepted*. On the other hand, "uphill" moves that increase the cost function are *rejected* and reversed.

When visualizing iterative improvement algorithms, a key goal is to demonstrate how a placement changes as moves are applied. This is accomplished by extending the placement view described in the previous section to display the "interesting events" of iterative improvement: *selection* of a cell and move, *application* of the move, and either *acceptance* or *rejection* of the move.

Fig. 3 shows how these events are displayed in sequence for a "Move-H" move that translates a cell horizontally. Before the move is applied, the selected cell is highlighted in orange, and an arrow is drawn originating from the center of the cell and ending at the point where the cell will be moved. At the same time, the name of the move and the amount of the offset are displayed in text at the bottom of the display. After the move is applied, the cell is shown in its new position at the

TABLE I
TYPICAL MOVES FOR ITERATIVE IMPROVEMENT

Move	Effect
Move-H	Translate selected cell horizontally by random amount
Move-V	Translate selected cell vertically by random amount
Rotate	Rotate selected cell 90 degrees
Flip-H	Flip selected cell horizontally
Flip-V	Flip selected cell vertically

end of the arrow, and text at the bottom of the view again indicates the name of the move and the offset, as well as the change in the cost function (dCost) and whether the move was accepted or rejected. If the move is accepted, the selected cell is highlighted in green, while if the move is rejected, it is highlighted in red.

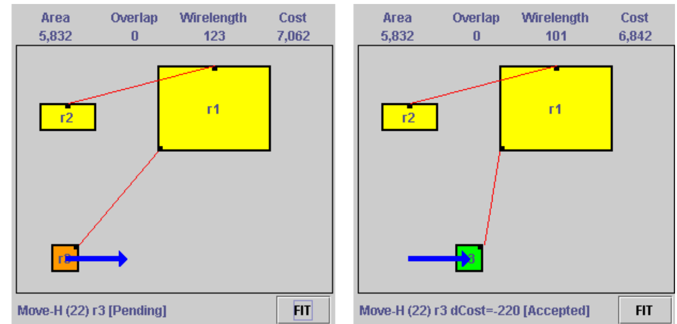


Fig. 3. Move Application during iterative improvement. In this case, the move decreases the cost function and is accepted.

Single-move animation is useful for viewing short sequences of moves, but iterative algorithms can apply moves hundreds or thousands of times. A separate *move history view* summarizes a sequence of move attempts, as shown in Fig. 4. The move history view plots the cost of the placement on a log scale. Each move attempt is depicted as a vertical bar that represents the current cost. Accepted moves are shaded green, while rejected moves are shaded white. A small red bar above each rejected move indicates the increase in cost that would have occurred had the move been accepted.

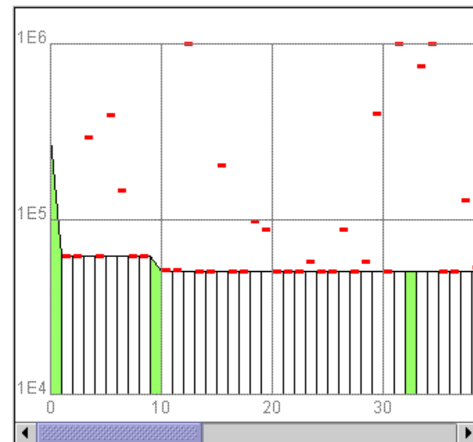


Fig. 4. Move History Display for Iterative Improvement. Each vertical bar represents one move attempt. Accepted attempts show a cost reduction and are shaded green. Rejected moves are shaded white; the red bar above indicates the rejected cost increase.

B. *Placement by Simulated Annealing*

Straightforward iterative improvement algorithms accept only downhill moves. While simple to implement, a drawback is that such algorithms may become trapped in a local minimum of the cost function. Simulated Annealing [14], [15] is a popular optimization scheme that accepts uphill moves in a controlled fashion, allowing the search to "escape" local minima.

Following the general framework of iterative algorithms, simulated annealing repeatedly applies moves to different cells in a placement, and downhill moves that decrease the cost function are always accepted. Uphill moves are accepted probabilistically under the control of a *temperature* parameter T. Specifically, a move that increases the cost function by ΔC is accepted with probability $P=e^{-(\Delta C/T)}$.

The algorithm operates in two nested loops. In the outer loop, the value of T is gradually lowered from an initial value (that allows most uphill moves to be accepted) down to a final value (where no uphill moves are accepted). In the inner loop, a large number of moves are attempted at each temperature value. Each time a move is attempted, P is calculated and compared to a random number r $(0 < r < 1)$. If $P > r$, the move is accepted, otherwise it is rejected and reversed.

To animate Simulated Annealing, the placement view described in the previous section is used as one view of the algorithm's operation, as shown in Fig. 5. A second view is used to explore how placement cost varies as a function of temperature – the classic "annealing curve" [14]. This display plots the minimum, maximum, and average cost placement accepted at each temperature versus temperature. "VCR controls" at the bottom of the display allow the user to start, stop, pause, and single-step the animation.

While simulated annealing is conceptually simple, there are a number of issues that make creating an effective implementation difficult [15], including the determination of initial temperature, number of move attempts per temperature, stopping criteria, and weighting of different types of moves. The "options" control button brings up a popup menu (not shown) that allows the user to experiment with several of these parameters, and also to adjust the animation speed.

The basic animation provided by this visualization allows the user to view the effect of individual moves. However, like iterative improvement, hundreds or thousands of moves are applied during simulated annealing. Thus the Move History view developed for iterative improvement has been extended to illustrate move history during simulated annealing, as shown in Fig. 6.

As before, a cost plot illustrates the cost impact and acceptance or rejection status of each move attempt. However, this plot must be extended to show accepted uphill moves. Thus downhill and uphill accepted moves are shaded in contrasting colors and brightness Another extension to the move history display illustrates the operation of the Metropolis Algorithm for each move attempt. Specifically, a lower display shows the probability P of each move and the random number r used to make the move acceptance decision for uphill moves. Again contrasting shading is used to illustrate which moves are accepted and rejected.

The simulated annealing visualization allows for animation in either a fine-grain or coarse-grain mode that can be selected from the options menu. In the fine-grain mode, each move attempt is displayed following the conventions described in the previous section. This gives the user a good idea of how individual moves are attempted, accepted, and rejected at each temperature. Single-stepping the visualization shows each move selection, application, and as appropriate its acceptance or rejection.

However, since even small placement problems require hundreds or thousands of move attempts at each temperature, the fine-grain mode is not appropriate for visualizing the overall operation of the algorithm. Thus the coarse-grain mode updates the display once at the end of each temperature instead of after every move. This allows the user to visualize the change in placement at each temperature. Single-stepping the visualization in coarse-grain mode moves forward an entire temperature.

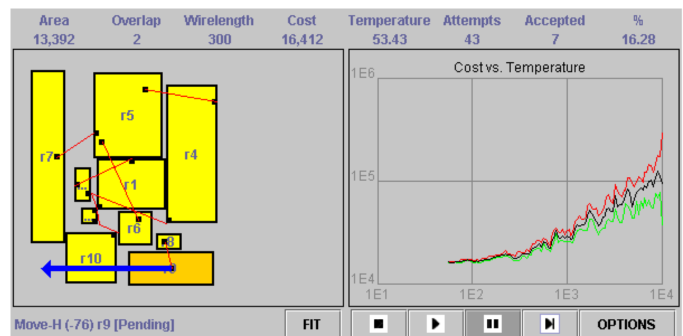


Fig. 5. Full visualization of placement using Simulated Annealing. In fine-grain mode, the display is updated after every move application. In coarse-grain mode, it is updated only at the end of each temperature.

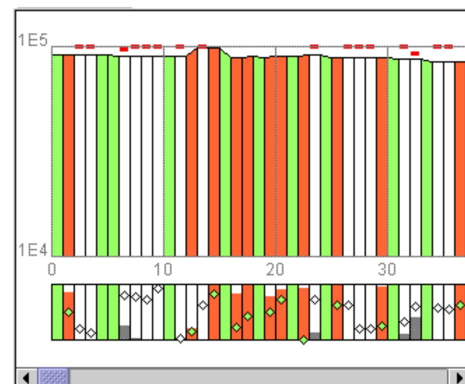


Fig. 6 – Move History Display for Simulated Annealing. The upper part of the display is similar to Fig. 4 except that accepted uphill moves are shaded in orange. The lower part of the display shows the acceptance probability P as a shaded bar and the random number r as a diamond-shaped marker.

IV. VISUALIZING ROUTING

The goal of routing is to find connections for nets between the terminals of cells. Routing is performed after placement, and when completed specifies the layout of all wiring necessary to implement the design. Two common approaches to routing are known as maze routing and channel routing.

A. *Maze Routing*

Maze routing models the routing surface as a grid in which each grid point can be a terminal of a desired connection (known as either the *source* or *target*), a *wire* that connects adjacent grid points, or an *obstacle* that represents space that is not available for interconnections. The grid is described by a two dimensional array which records the state of each grid point

Many approaches to maze routing have been attempted over the years, but the Lee Algorithm [16] remains popular because it is guaranteed to find a shortest-path connection if one exists. It operates in two phases. During the *expansion* phase, the algorithm first marks the *source* and *target* endpoints of a desired connection, and then searches outward from the source while labeling each node with its distance from the source. When the target is reached, the *backtrace* phase selects a path that follows decreasing label values and marks these as wires. These wires act as obstacles for later routings.

Although the Lee Algorithm operates using only one layer of interconnect in its basic form, it can be easily modified to support multiple layers expanding the two dimensional grid into three dimensions. The expansion and backtrace phases then proceed in both horizontal and vertical directions.

Several other extensions to the algorithm have been explored over the years, including the use of directional labels to reduce the storage required for each grid point and the use of non-uniform costs to bias routing to avoid particular regions of the routing surface or particular directions of routing.

To construct a visualization of the Lee Algorithm, it is first necessary to create a graphical representation of the problem formulation i.e., the routing grid. This can be done in straightforward fashion as shown in Fig. 7. Each point is represented by a square in the grid that is shaded and labeled to reflect how it changes in state as the algorithm executes. Next, a set of interesting events must be defined, along with their effect on the graphical representation.

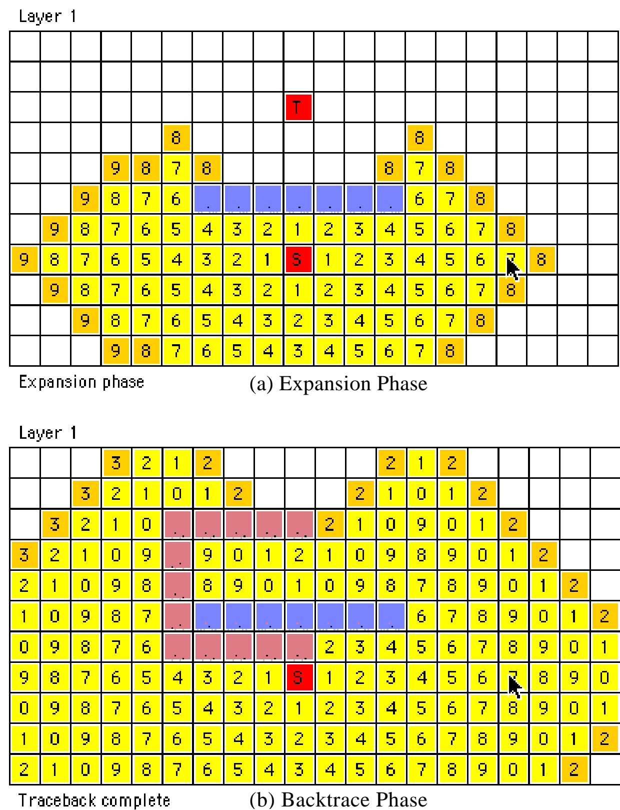
During the expansion phase, interesting events include: 1) definition of the source and target terminals (labeled “S” and “T” and shaded red); 2) initial labeling of grid points as the algorithm places nodes in a queue (labeled with distance from source and shaded orange), and removal of each node from the queue as expansion passes to the grid point’s unlabeled neighbors (change shading from orange to yellow). Fig. 7 (a) shows a snapshot of the visualization during expansion. Note that the labeling shows how the search expands in a diamond-shaped “wavefront” until the target is reached.

During the backtrace phase, interesting events 1) include the labeling of grid points as obstacles as a path is selected (shown as colored cells); and 2), and the removal of labels after a complete connection is found. Fig. 7 (b) shows a snapshot of backtrace phase as it nears completion.

In the Lee Algorithm visualization the user specifies source and target terminals by clicking the mouse over individual gridpoints. The expansion and backtrace phases are then displayed as described above. The applet is parameterized to allow the display of multiple layers.

B. *Channel Routing*

A major drawback of maze routing is its computational complexity. One way to reduce this complexity is to constrain the routing problem. In the *channel routing* problem formulation, connection terminals must be placed in *columns* at the top and bottom of the routing region (channel). Horizontal connections between terminals are assigned to *tracks* in the channel. Vertical connections occupy columns on a different layer so that they pass over unconnected layers in tracks, while *vias* connect the horizontal and vertical connections of connected nets.



(a) Expansion Phase

(b) Backtrace Phase

Fig. 7. Maze Routing Visualization. The expansion phase (a) displays the search for a path from the source (S) and target (T) while avoiding an obstacle. The backtrace phase (b) marks the shortest path connection. Distance labels are truncated to one character for readability.

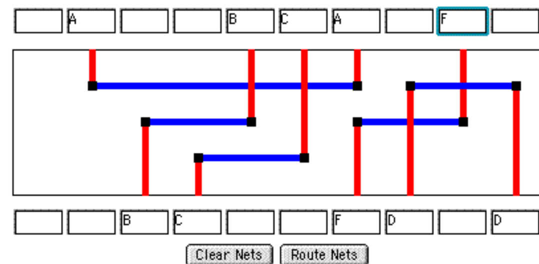


Fig. 8. Channel Router Visualization. Net names are entered into the fields representing terminals at the top and bottom of the channel. This example includes a vertical constraint between nets “A” and “F”.

While constraining the routing problem in this manner can have a negative impact on design quality, it greatly reduces the complexity of the problem. The Left Edge Algorithm (LEA) [17] can find connections by sorting nets by their left

edge and then assigning each net in that order to the first track which contains no overlapping nets. The algorithm is complicated by the possible presence of *vertical constraints* – if different terminals are placed at the top and bottom of the same column, the net connected to the top terminal must be routed above the net connected to the bottom terminal. Constraint relationships form a *vertical constraint graph* (VCG); VCGs containing cycles cannot be routed by the LEA without inserting a *dogleg* [2].

Fig. 8 shows the visualization of the channel routing problem formulation and the LEA. Connection terminals are displayed as fields at the top and bottom of the routing channel; these can be entered directly by the user. Interesting events include the selection of connections for placement in the channel and the assignment of each connection to a track.

V. Results

Each of the CAD algorithm visualizations has been implemented as a Java applet. The routing applets were developed first using the original Java Abstract Windowing Toolkit (AWT). The placement applets (iterative improvement and simulated annealing) were developed later and took advantage of the more capable Swing user interface library [18]. The applets range in size from 500 lines of code for the Channel Router Applet to 4500 lines for the Iterative Improvement and Simulated Annealing placement applets, which are implemented together. Compiled code is relatively small, ranging from about 10-80Kbytes in a JAR (Java Archive) file.

The applets have been used as a teaching aid during lectures in VLSI Design courses at Lafayette College. Each applet is used to demonstrate the basic formulation and is be rerun with different input data and conditions to illustrate important points of each algorithm. Student response has been positive.

The applets have been made available on the world-wide web at http://foghorn.cadlab.lafayette.edu/cadapplets/. This site has been indexed by several search engines and receives considerable traffic. To gain some insight into where these accesses were coming from and which applets were being run, we examined the web server's log file for the over the period from January 2001 - February 2003. To differentiate between casual visitors actually running the applets, we focused on number of times each applet was downloaded and run by counting only accesses to compiled JAR files.

Table II shows the results of this analysis, broken down by applet. The "Placement Formulation Demo", illustrates the placement user interface and allows the user to manipulate a placement by dragging modules. The "Simulated Annealing" applet performs simulated annealing as described earlier but did not include the Move History Display, which had not yet been developed. The "Maze Routing" and "Channel Routing" applets operate as described in the previous section.

Table II also differentiates between accesses from local (on-campus) vs. non-local (off-campus) sources. Non-local accesses came primarily from universities, semiconductor manufacturers, CAD tool developers, and individuals.

TABLE II
Applet Access Summary

Applet	Local Accesses	Non-Local Accesses
Placement Formulation Demo	10	324
Simulated Annealing	15	464
Maze Routing	95	1477
Channel Routing	75	1020
Total	195	3285

VI. Conclusion

This paper has discussed the use of software visualization techniques to develop animations of common VLSI CAD algorithms. The applets have been used successfully as an aid to instruction in VLSI Design courses and have been downloaded extensively via the Internet.

There are a number of areas for future work in this project. First of all, the visualizations developed so far provide only a small subset of the approaches used in VLSI CAD – many more are needed for thorough coverage. Existing visualizations should be extended to include considerations such as timing. Visualization should be applied to larger and more realistic examples. Finally, a more formal evaluation of the effectiveness of the visualizations should be performed.

References

[1] M. Saraffzadeh and C. Wong, *An Introduction to VLSI Physical Design*, McGraw-Hill, 1996.

[2] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd. ed., Kluwer Academic Publishers, 1999.

[3] J. Nestor, "Web-Based Visualization Tools for Teaching VLSI CAD Algorithms", *Proceedings International Conference on Microelectronic Systems Education*, pp. 100-101, June 2001.

[4] J. Nestor, "Animation of VLSI CAD Algorithms – A Case Study", *Proceedings of the ASEE Annual Conference*, June 2002.

[5] M. Brown and R. Sedgewick, pp. 28-39, "Techniques for Algorithm Animation", *IEEE Software*, Vol. 2, No. 1, January 1985.

[6] J. Stasko, J. Domingue, M. Brown, and B. Price, ed., *Software Visualization: Programming as a Multimedia Experience.* MIT Press, 1998.

[7] K. Arnold, J. Gosling, and D. Holms, *The Java Programming Language, 3rd Edition*, Addison-Wesley, 2000.

[8] M. Brown, R. Raisamo, and M. Najork, "A Java-based implementation of Collaborative Active Textbooks", *Proceedings of the IEEE Symposium on Visual Languages,* September 1997.

[9] I. Moon, "Action Based Learning for Switching and Automata Theory", http://vlsi.colorado.edu/~mooni/N_ABLE/N_ABLE.html.

[10] R. Drechsler, "JADE: Implementation and Visualization of a BDD Package in JAVA", In *Proc. DATE*, page 259, 2002.

[11] R. Hentschke and R. Reis, "Blue Macaw Didactic Placement Tool", http://www.inf.ufrgs.br/~renato/bluemacaw/.

[12] R. Rutenbar, "18-760 Projects", http://www.ece.cmu.edu/~ee760/760projects.html.

[13] Szollar, S and Young, J., "The Incredible Anneal-O-Matic", http://www-cad.eecs.berkeley.edu/~jimy/classes/ee244/hw2/index.htm.

[14] S. Kirkpatrick et. al., "Optimization by Simulated Annealing, *Science*, Vol. 220, No. 2298, pp. 671-680, 1983.

[15] R. Rutenbar, "Simulated Annealing Algorithms: An Overview", *IEEE Circuits and Devices Magazine*, pp. 19-26, January 1989.

[16] C. Y. Lee, "An Algorithm for Path Connections and its Applications", *IRE Transactions on Computers*, vol.EC-10, pp. 346-365, 1961.

[17] A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment Within Large Apertures", *Proc. DAC*, pp. 165-173, 1971.

[18] K. Walrath and M. Campione, *The JFC Swing Tutorial*, Addison-Wesley, 1999.