Multithreading in the Electric VLSI Design System

Steven M. Rubin

SML 2006-0316
Sun Microsystems Laboratories
September 17, 2008

Abstract

The Electric VLSI Design System is an open-source circuit-design system that has been used for decades to make integrated circuit (IC) chips. Written in 1982, it predates most commercial circuit design systems available today. Beginning in 2003, a small team at Sun Microsystems translated Electric from C to Java, completing the task in less than two years. The resulting system is more stable, has a much improved user interface, and (to the surprise of many) is faster.

One of the reasons for the new translation was to take advantage of Java's powerful multithreading facilities. While attempting to make use of these facilities, it was determined that a thread-safe database was needed. We split the system into a database server and a user-interface client. This new database has a number of advantages, including: collaborative design, thin-client design terminals, reliable crash recovery, reduced memory usage, and the ability to use multiple processors.

This paper describes these improvements to Electric and presents two multithreading facilities that have been built: a design-rule checker and a wire router.

Introduction

The Electric VLSI Design System has been used to do circuit design for over 25 years. Originally developed at the Fairchild/Schlumberger Palo Alto research laboratory [Rubin], Electric has been available since 1983 in various forms (see Figure 1). Over the years, Electric has developed an international reputation. Both Canada and New Zealand created institutions to support Electric for universities in their countries. Today there are many universities using Electric as well as hobbyists and corporate designers. The system has been used to design thousands of IC chips.

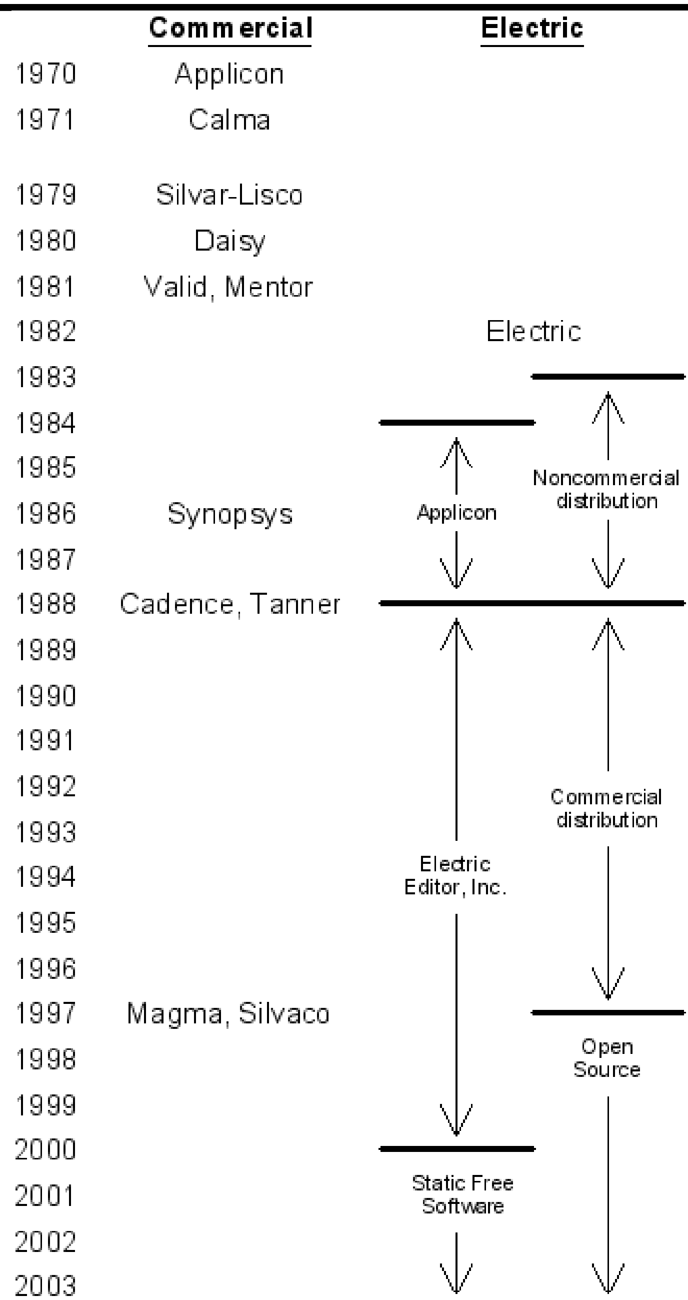

Figure 1: The History of Electric and the EDA World
Electric has been distributed in many different ways. From 1983 to 1988, it was available for noncommercial use; from 1988 to 1997 it was sold commercially; and since 1997 it has been open-source.

Over the past 25 years, integrated circuits have grown increasingly complex, making many demands on the computer-aided design (CAD) software. The two major demands are for *size* and *complexity*. The demand for circuit size is clear from Moore's Law [Moore] which describes the growth of computers, and therefore of IC chips. The demand for complexity is due to the ever-changing processes used in IC chip production. This race to build the smallest possible transistor forces design software to provide many new analysis and synthesis tools. For example, the 3D nature of wires on different layers has become more important recently, and many tools have been created to measure inductance, capacitance, and other parasitic effects.

Electric addresses both the issues of size and complexity. It manages the ever-growing size of circuits in the same way as all other CAD systems: through the use of hierarchy, improved algorithms, and fine-tuning. Recent trends in computing favor powerful servers with "thin client" design terminals, and Electric even makes use of these capabilities. Nevertheless, CAD algorithms are some of the most resource-intensive systems, and Electric fares no better or worse than any other.

Electric is particularly good at handling the problem of circuit complexity. Most CAD systems are intricate collections of files and filters, weaving disparate pieces of software into a delicate web. Electric is a single program with a common database, so tools can easily share information and can make use of common facilities to simplify their work. The Electric database has a common interface for all design operations (called *tools*). Tools can schedule *jobs* to interact with the database, and can also create *listeners* that watch database activity, thus enabling them to run interactively. Essentially, an Electric tool is treated in the same way as an operating system "task".

In addition to the seamless integration of tools, the Electric database is extensible, allowing individual tools to store private information, which is retained as a persistent part of the database. The tool flexibility and database extensibility have provided a powerful environment for CAD tool development, and in the system's 25 years of existence, every new CAD function has been easy to integrate.

Besides easy integration of tools, Electric also provides an elegant way of incorporating different environments of design. Each fabrication process is encapsulated in a *technology*, which describes the geometry, topology, and behavior of that foundry. For example, the MOSIS CMOS technology describes design rules (geometry), layer interconnect rules (topology), and Spice deck headers (behavior). Electric also supports non-layout design environments, including Schematic capture and Hardware Description Languages (HDLs).

Viewed abstractly, the task of circuit design is really a two-dimensional space, where one axis lists the different operations that can be performed on circuitry, and the other axis lists the different environments of design (see Figure 2). This space is really a sparse matrix, populated by the ever-growing list of synthesis and analysis tools. By providing an elegant way of populating the matrix, Electric offers flexibility that is unavailable in other tools.

	CMOS	BiPolar	Schematics	Artwork	FPGAs	Architecture	HDL
DRC	X	X	X				
ERC	X	X					
Parasitics	X	X					
Simulation (low level)	X	X	X				
Simulation (high level)			X		X	X	X
Placement					X		X
Routing	X	X			X		X
Silicon Compilation			X			X	X
Network Consistency Checking	X	X	X			X	
Logical Effort	X		X				
HDL Compiler							X

Figure 2: The Space of Design

Electric's Unique Style of Design

There are two basic user-interface styles that a graphical CAD system can offer: *connectivity* and *geometry*.

- The connectivity approach is used by every schematic capture system: components are placed, and wires are drawn to connect them. Because the system understands that the wires connect to the components, the circuit remains properly connected, even when components are moved.
- The geometry approach is used by most integrated circuit layout systems: areas of "paint" are laid down on different layers to form the masks for chip fabrication. There is no understanding of connectivity, and the movement of pieces of paint can alter the topology of the circuit. Also, and any needed connectivity information must be extracted from the geometry in a costly and error-prone "node extraction" phase.

Electric is different because it uses connectivity for all design, even integrated circuit layout. This means that the designer places components (MOS transistors, contacts, etc.) and draws wires (metal-2, polysilicon, etc.) to connect them. The screen shows the true geometry, but the system also knows the connectivity.

Electric uses graph terms for its style of design, referring to the components as *nodes* and the wires as *arcs*. Nodes have *ports* on them, which are the sites of arc connections (see Figure 3). Arcs always connect exactly two nodes. If the designer wants a wire to follow a serpentine path, it must consist of multiple arcs, connected by intermediate "pin" nodes along the way. The nodes are arcs are grouped into *cells*, and cells are grouped into *libraries*.

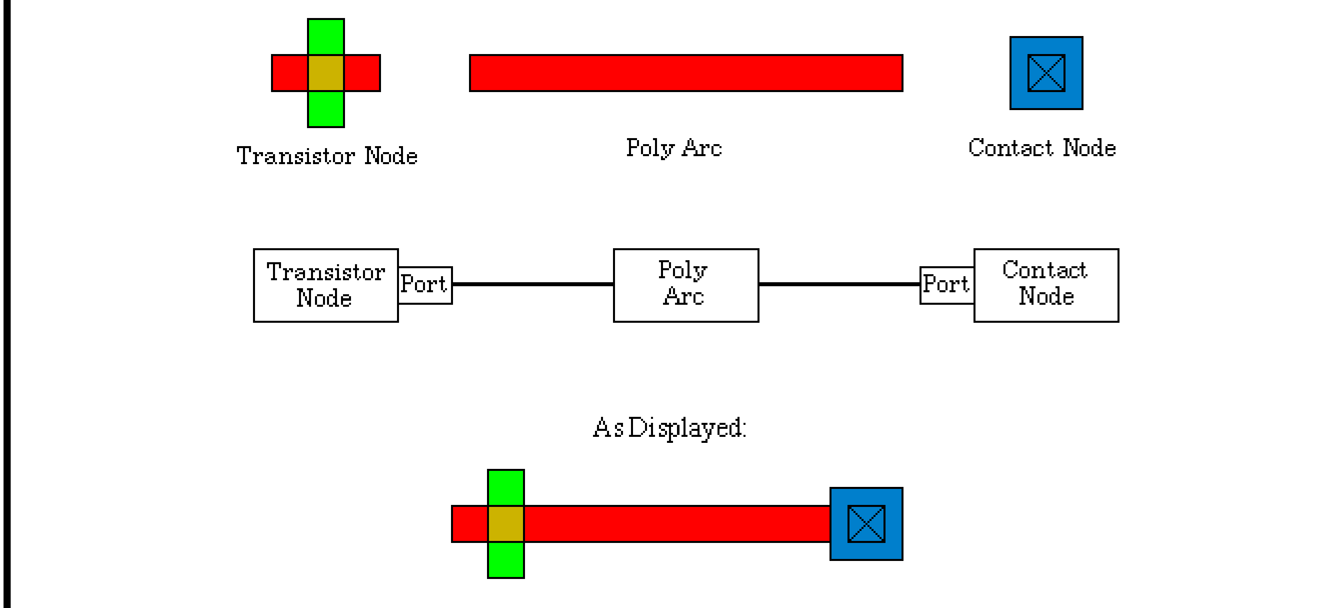

Figure 3: Nodes, Arcs, and Ports in a Cell

The nodes-and-arcs concept is also used to implement hierarchy. Nodes can be *primitive*, such as MOS transistors and contacts, or they can be *complex* which means that they are instances of other cells. The definition of a node, therefore, depends on the type of the node: primitive nodes are defined by technologies, and complex nodes are defined by cells in libraries.

Connectivity of the circuit is also implemented in a hierarchical manner, by having ports on both types of nodes: primitive and complex. The ports on primitive nodes are established when the primitive nodes are defined in the technologies, but ports on cell instances are specified by the user. By taking an existing port of a node, and *exporting* it to the outside world, that location is now available as a port on instances of the cell.

One of the most powerful features of connectivity-based layout is that constraints can be placed on the connections. Constraints tie different parts of the circuit together, and react to changes by making other, necessary changes. In a typical schematics system, constraints are predefined and never change. The system has a fixed set of rules for deciding how to keep the layout consistent. In Electric, constraints are user-programmable. Although Electric establishes a default set of constraints that make sense in normal use, the user can override these constraints to "program" the layout appropriately.

For example, assume that a power rail runs horizontally, with vertical drops to the various components that need power (see Figure 4). If the rail is to be moved up, it might normally be necessary to move each horizontal segment, and extend the vertical ones. With Electric's constraint system, each horizontal segment is given the *fixed angle* property, which requests that the arc remain at the same angle (horizontal). Then, if one end of the power rail is moved up, the segment responds by pulling the other end up to keep the arc horizontal. This ripples down the power rail and moves it all up at once. The vertical drops then stretch to connect to the power rail.

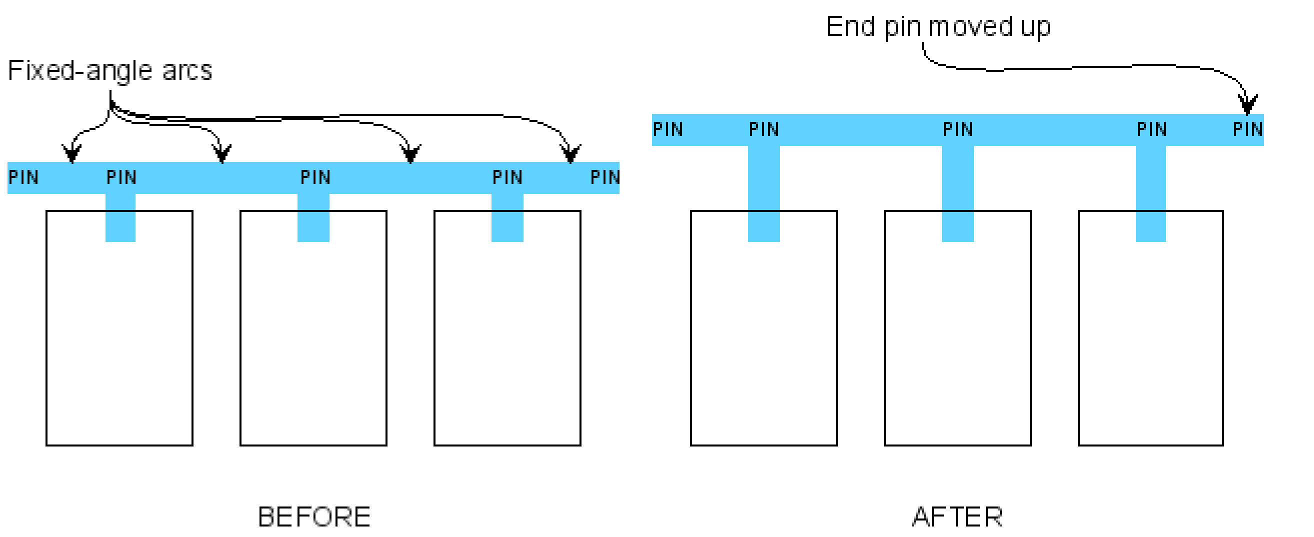

Figure 4: Fixed-angle Constraints

Electric solves constraints within a cell by propagating changes from node to arc. In addition, Electric propagates constraints up the hierarchy by examining cell instances when their contents have changed. For example, if a memory cell is designed with a given pitch, and an array of these cells is placed and connected using that pitch, then it can be very hard to change the size of a single cell without damaging the upper-level instances of that cell. In Electric, it is only necessary to place the *rigid* constraint on the arcs that connect the cell instances. Then, by moving the export inside of the memory cell, it will expand or contract the cell, and the array will adjust accordingly (see Figure 5).

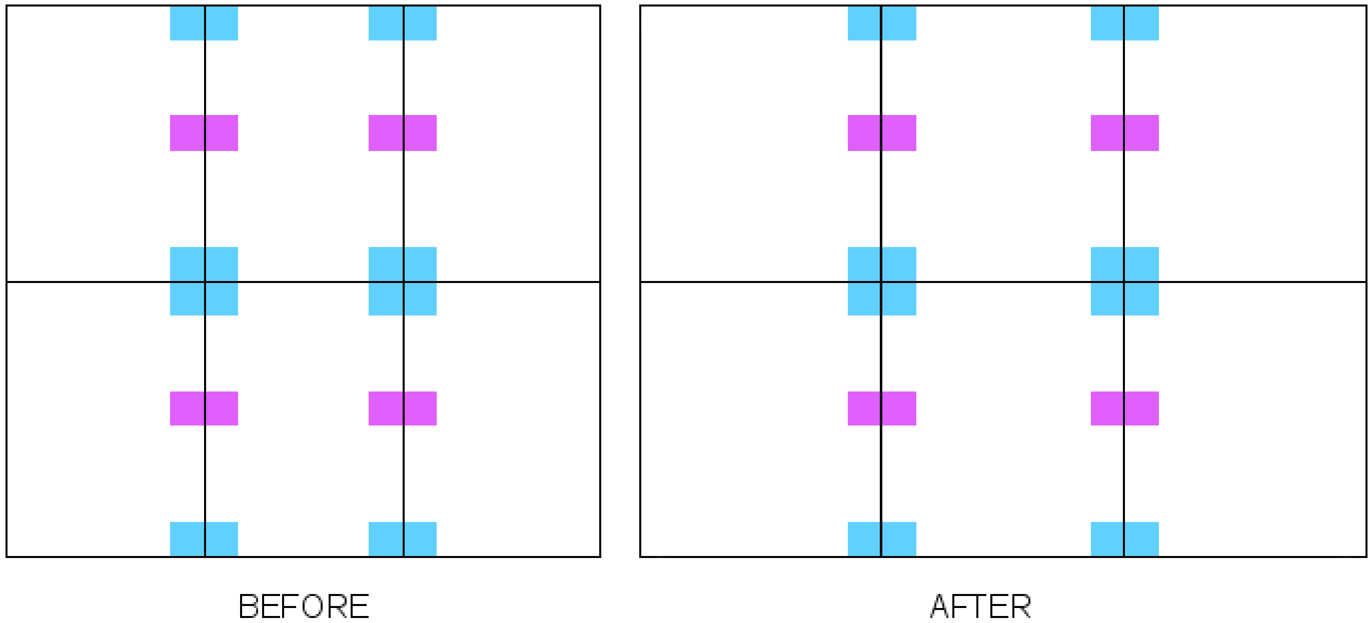

Figure 5: Rigid Constraints in a Hierarchical Design

In summary, there are many advantages to the use of connectivity-based IC layout:

- **No node extraction.** Geometry-based systems must go through a "node extraction" process to convert layout into connected components whenever such connectivity information is needed. In Electric, node extraction is not a separate, error-prone step. Instead, the connectivity is part of the layout description and is instantly available. This speeds up all network-oriented operations, including simulation, layout-versus schematic (LVS), and electrical rules checkers.
- **No geometry errors.** Complex components are no longer composed of unrelated pieces of geometry that can be moved independently. In geometry-based systems, you can select each layer of a complex component independently. For example, if you accidentally move the polysilicon part of a transistor to a different location, it will cease to be a transistor. In Electric, the transistor is a single component, and cannot be accidentally destroyed.
- **More powerful editing.** Browsing the circuit is more powerful because the editor can show network information whenever appropriate (such as during selection). Also, Electric combines the connectivity with a layout constraint system to give the editor powerful manipulation tools. These tools keep the design properly connected, even as the circuit is modified on different levels of hierarchy.
- **Tools are smarter** when they can use connectivity information. For example, the design rule checker knows when two pieces of layout are connected and uses different spacing rules.
- **Simpler design process.** When doing schematics and layout at the same time, designers typically iterate between design-rule checking (DRC) and layout-versus schematic (LVS). The problem is that the LVS system needs to know the circuit connectivity, which is

derived through node extraction, which can only run if the layout is DRC-clean. So first the layout must be DRC clean before running LVS. Then, when LVS problems are found, the layout must be fixed and made DRC clean again. Much time is wasted getting the layout to be DRC-clean. Since Electric can extract connectivity for LVS without having perfect design rules, an Electric user's first step is to get the layout and schematics to match. Then the design rules can be cleaned-up without fear of losing the LVS match.

- **Common user interface.** By using the same style of interaction for schematics and IC layout, a single CAD system with a single user interface can be used to do both. Electric incorporates the notion of different "views" of a circuit (schematics, layout, HDL, documentation, etc.) so that it can handle all aspects of circuit design.

The disadvantages of connectivity-based IC layout are also known:

- **It is different** from all the rest and requires retraining. This is true, but many have converted and found it worthwhile.
- **Requires extra work** on the user's part to enter the connectivity as well as the geometry. While this may be true in the initial phases of design, it is not true overall. This is because the use of connectivity, early in the design, helps the system to find problems later on. In addition, Electric has many "power" tools that automatically create the connectivity information.
- **Design is not WYSIWYG** (what-you-see-is-what-you-get) because objects that touch on the screen may or may not be truly connected. Electric has many tools to ensure that the connectivity has been properly constructed.

Translating Electric from C to Java

In September of 2003, a small team began to translate Electric from C/C++ into Java. The reasons for this translation were numerous, including:

- The allure of platform-independence
- The promise of improved stability
- The opportunity to better manage multithreading
- The opportunity to rethink 20 years of design decisions

The driving reason was the first one: platform independence. As a C program, Electric achieved this goal by having interface modules for many operating systems (Macintosh OS 9, Windows MFC, UNIX/Linux Motif widgets, and Qt). With the arrival of Macintosh OS X, it was simply easier to deal with Java Swing than yet another computer-specific interface package.

We had many concerns about translating Electric, mostly based around the fact that we were abandoning a mature and working system. We were also concerned about efficiency. Java was not known for being time or space efficient, and CAD systems such as Electric demand as much as they can from a computer in order to accomplish complex tasks.

In 10 months, enough of the system was translated to enable designers to use the new code. After two years, the entire system was translated into Java. Electric now has a more modern interface. It is much more stable, due in part to our extensive use of threads (every command is run in a separate thread, so even if experimental code fails, only that thread is lost, and the main system continues to run). What surprised us the most is that the code is also faster.

Speedup of Electric code was achieved through better algorithms. Java makes these algorithms much easier to code by providing extensive class libraries, powerful memory management, and ease of debugging. Two examples illustrate this result: *full-chip redisplay* and *layout-versus-schematic* (LVS).

Designers often request full-chip redisplay to get the "big picture" of a circuit. These full-chip images do not offer any useful detailed information, because the individual transistors and wires are too tiny to see on the screen. Nevertheless, the full view does show overall structure, and so this operation is done fairly often. The C version of Electric could draw a one-million-transistor chip in about a minute (which is too long). The Java version of Electric has a number of speedups that reduce this time to a few seconds. In addition, C Electric could not pan the chip (shifting left/right/up/down) without complete redisplay, whereas Java Electric can pan smoothly. This has been accomplished primarily by caching cell contents so that already-rendered parts of the chip can be copied to similar places elsewhere on the screen.

One would think that a simple display-caching algorithm could have been implemented in the C code as well. But it would have been extremely difficult to do this because of C Electric's platform-independence. In order to achieve its platform-independence, C Electric defined an API that captured the essential features of an operating system. This API was used to draw graphics, popup a menu, present a dialog, etc. Electric then called its own internal routines to perform these tasks. Each different operating-system module then implemented those routines for that platform. Unfortunately, the routines available did not permit a powerful combination of rendering and caching. Although such a facility could have been added, it would have been an enormous development task because it would have had to be implemented six times for every supported platform. In 20 years of development, the idea was considered many times, but the effort was simply too vast to undertake. The Java implementation of this idea was completed in one week (by one person).

Another example of the speedup that we obtained by translating Electric from C to Java is in the area of layout-versus-schematic, or LVS. LVS takes a layout and a schematic version of a circuit and compares them to make sure that the layout is correct. In the C version of Electric, this task was very slow, often taking hours to check a full chip. In Java, the code was completely rewritten (without examining the old C code). The new system runs over 100 times faster, thanks to a combination of clever algorithms and the use of hierarchy.

A Thread-Safe Database

When we started to build Electric in Java, we decided to run every command in its own thread, thinking that this would enable massively parallel computation. Our initial implementation consisted of a "readers and writers" scheme in which jobs that change the database must run alone, but jobs that merely examine the database can run in parallel. However, there are many complicating issues involved here, such as the notion that redisplay (an examination job) should be able to run at any time, even during a change job. To prevent concurrency problems, we had to run each thread separately, which eliminated any advantage that multiple threads might offer.

The solution to this problem was to build a thread-safe database. The essential goal of such a database is to provide every running thread with its own private *snapshot* of the circuit, so that even if another thread changes the database, the current thread's data is safe. Examination jobs can access their copy of the database for as long as they like, even during changes, without fear of concurrent modification problems, because each snapshot is *immutable* and never changes. Change jobs transform the database into a new one, and create a new snapshot that includes the change. Undo and Redo simply consist of selecting a different snapshot from an earlier or later time.

Given this database model, the system can run as many parallel jobs as it likes. Of course, multiple change jobs can cause conflicts, so the database must check to be sure that each new snapshot is valid. Changes to the main database are only accepted when the change job completes and the new snapshot is free of conflicts. If the change job crashes or the snapshot is inconsistent with other changes that have been made, the snapshot is rejected and the database is unchanged.

Realistically, it is not possible to create multiple copies of the circuit database for every job. In fact, the circuit database is so large that there is often barely enough room for just one copy. Instead, the immutable database must share data wherever possible, duplicating only the parts of the database that have changed. It may seem as if retaining all of these snapshots will use an excessive amount of memory, but in fact, the costs are recovered by the fact that an undo/redo system is no longer needed. In fact, the cost of the extra snapshots is comparable to the cost associated with tracking changes for undo/redo.

Typical databases are implemented on very simple tabular data structures with only a few different data types (numbers and strings). Electric has a much more complex data structure, involving many different object types (nodes and arcs, instances and prototypes, libraries and technologies, etc.) The system also has complex interconnections between data entries due to its use of hierarchy, the different representations (geometry, topology, behavior), and much more. This is the main reason that Electric has always had a custom database, rather than making use of existing database software. The Electric immutable database also had to be build from scratch.

Implementation of the Electric database was made simpler by our use of Java "collection" classes. In the C version of Electric, all lists were linked-lists that started from a header and wound through every object in the list. Changes to a list required changes to pointers on individual elements of the list. In Java Electric, lists are not linked, they are just arrays which are stored at the header location. No data is stored on the individual elements. Thus, a change to a list involves making a new copy of that list, but it does not affect the elements in the list.

This feature allows copies of the database to share most data (the elements) while making changes only to the lists. For example, assume that a user places a new transistor inside of a cell. This cell is inside of a library of cells, and the library is in a list of libraries that comprise the database snapshot. When the new transistor is created, a new copy of the cell object is created. This new copy shares all data with the old cell except for the list of nodes, which now has an extra entry. The new cell is placed in a new Library, which shares all data with the old library except for its list of cells, which now includes the new cell with the new node list. Finally, the new snapshot shares all information with the old snapshot except for a changed list of libraries. Thus, the difference between the old and new database is the duplication of three skeletal objects (a cell, a library, and a snapshot) and three lists (in the snapshot, library, and cell).

Modification of an object requires that a new copy be created which shares all values except for those fields that have changed. This new object replaces the original one in the cell/library/snapshot. Deletion is simply the adjustment of lists to remove a pointer. More complex changes to a cell (that might involve many objects) will not demand much more in the way of memory usage.

In order to guarantee that each snapshot is valid, it is necessary that no field in any object ever change. Rather than change anything in an object, a new copy of the object is created. In the new object, new data values are created for the changed fields, and shared pointers are used to fill in all fields that did not change. All of the unchanged fields can safely share their data, because those values never change, either. This is why the database is called "immutable".

One advantage of using Java to build the immutable database is that the language has constructs for guaranteeing immutability. The "final" keyword tells the system to prevent any changes to fields in the object. Powerful memory management automatically destroys unused objects when their snapshots are abandoned and they are no longer referenced. Thus, the developers can rely on these formal properties of the immutable database.

There are four properties of a good database, called the ACID test (Atomic, Consistent, Isolated, Durable) [Gray]. Electric's database is a "relaxed ACID" system that fails to fully offer one of these properties. This list shows how the system fares in providing these important database requirements:

- **Atomic (changes to the database are all-or-nothing).** Electric meets this requirement. Each snapshot is a complete database, and any problems encountered during a change merely roll the state back to the last valid snapshot.
- **Consistent (database is always valid).** Electric meets this requirement. Change routines check parameters and ensure a valid database.
- **Isolated (multiple parallel transactions do not impact each other).** Electric relaxes this requirement. Since changes may arrive at the database in arbitrary order, if they have been invalidated by previous changes, then they will fail. For example, if two change jobs run at the same time, one moving a component and the other deleting that component, then their order of execution is important. If the deletion runs first, then the movement will fail. The database detects these conflicts, and the deletion job will be rejected.
- **Durable (database changes are not lost).** Electric meets this requirement. This is accomplished by saving the serialized byte-streams that describe database change. The data can be replayed to recover from a crash.

Electric's new database offers a level of security that was unknown in the earlier C version of the system.

Further Database Improvements

To manage this new database, Electric was split into two parts: a database *server* and a user-interface *client*. The server manages the database, and runs all change jobs. The client interfaces to the user and schedules jobs. The client and server communicate with each other through sockets, and exchange objects by "serializing" them into a minimal byte stream of data. Client/server models have many advantages, including memory reduction, database security, and collaborative design.

One advantage of the client/server model is memory reduction. The server needs to retain only the essential parts of the database (those that are written to disk, for example). It does not need to maintain any user-interface data structures, which can be quite large. For example, the snapshot contains nodes and arcs, along with their location in the circuit. But to do rapid spatial search in the circuit, more sophisticated data structures are used. The client builds an R-tree [Guttman] from the node and arc information so that it can quickly locate objects at any point in the cell. A further memory-reduction is realized by having the client JVM hold only as much of the database as it needs to see, loaded on demand.

Examination jobs can run on either the client or the server. For example, redisplay runs on the client, enabling it to keep the display correct at all times. However, extensive examination jobs, such as design-rule checking, run on the server (or perhaps a second client). As we describe later, it is even possible to decompose these tasks into multiple processes, each examining a different part of the circuit and running on a different processor.

Change jobs always run on both the client and the server:

- The first part of a change job runs on the client and sets up the job. It examines the database and prepares the work that needs to be done on the server. This data is then sent to the server
- The second part of a change job performs the actual change activity, which runs on the server and creates a new snapshot. Results from this activity are sent back to the client, including a success indicator and any optional data that the third part needs.
- An optional third part runs on the client after the job has completed. It allows the user interface to do cleanup and display results.

Because the communication between the client and the server is done with sockets, and because the data is highly compacted by the Java serialization package, it is possible to run the Electric client remotely from the server. It is even possible to run multiple clients with a single server. For example, one user starts Electric (a server and a client communicating tightly on a single machine). Another user starts a client, but connects it to the first user's server. Since there is only one server, any change requested by either client will be visible on both. We have successfully run this "collaborative design" scenario over the greatest possible distances (one in San Francisco, the other in Moscow).

Another possible use of the client/server architecture is to create a single server process, and then have all design activity take place on "thin client" design terminals. This goes beyond collaborative design, and becomes a project management mechanism. It is particularly attractive in situations where the circuit requires many gigabytes of memory, and a powerful server machine is needed to manage the data. Since the clients typically use only a subset of the circuit, they do not need full copies of the data, and can run on smaller, less expensive machines. The server, no longer tied to a particular user's screen, can perform background operations such as database saving and circuit analysis.

One would think that the notion of thin client terminals for a large design project is a simple extension of the collaborative design scenario that has already been implemented. However, there are many complications. These complications are unimportant in a collaborative design scenario, where the two users are communicating (typically by phone) as they work together. However, when the users are not in touch with each other, and only communicate through their changes to the database, many problems can arise.

One problem is the notion of "undo", which the database accomplishes by reverting to an earlier snapshot. However, imagine this scenario: user A makes a change, then user B makes a change, then user A undoes their change. When this happens, the database will also undo user B's change since it occurred after the change that user A made. One suggested solution is to retain the serialized byte-streams that describe each change to the database. When user A undoes their change, the database is reverted, and then user B's change is "replayed". Of course, there are conflict-resolution issues that can arise from such an approach, but no more serious than those involved in handling multiple parallel change jobs.

Another problem involved in thin client design terminals has to do with preferences. Each user has a set of preferences that tailor their design session. Some affect only the user interface on the client, but some preferences also affect how the server will act. One such preference is the notion of "locking" an object so that it cannot be changed. Locking is useful in many phases of design activity, but each user wants to define their own set of locked objects. How is the server to know which objects are locked when each user has a different list? Keep in mind that the server may be in a remote location and know nothing about the user's environment.

Many solutions have been proposed for the problem of preferences. One solution involves sending preferences from the client to the server with each job. This could be a very large amount of data and it would slow the system. Another solution is to have the server ask the client for preference settings. However, this complicates the otherwise simple communication between the client and the server. Yet another solution is to have the server manage all preferences for each user, knowing which user requested a job and using the appropriate settings. This also complicates the server, and involves it in user-interface functions that it should not be doing. All of these solutions are less than optimal, and none has been chosen yet.

Multithreaded Tools

Using the improved database, a number of important CAD functions were decomposed so that they could make use of multiple processes and multiple processors. The design-rule checker and the wire router were reprogrammed, and both demonstrated improved performance. Also, a static timing tool was multithreaded, although there are no performance improvement results for it.

Design-rule checking is an intensive operation that examines all geometries on an integrated-circuit chip to ensure that they follow the rules for fabrication. These rules dictate size and spacing limitations for the many layers on a chip. In technologies that are 90nm or smaller, area and coverage calculations become extremely important and these tasks require expensive geometrical merging operations in order to do the necessary calculations. Although the hierarchical structure of a chip can be used to speedup some of this process, it is still very time-intensive, and must be repeated whenever a change is made to the circuit. Since design rules are classified by layer on the chip, there was a simple multithreading technique available: use one processor for each layer's rules. The Electric job manager was used to assign each thread to an available processor. The code was quickly implemented and produced a 1.5x speedup on a 4-processor machine. More tests are planned for larger systems, and more sophisticated partitioning, based on geometry, is expected to produce even better results.

Wire routing is the process of connecting the circuit together by placing wires on different layers of the integrated circuit. The process begins with a set of function blocks that are abstractly connected (i.e. the output of block 34 must connect to the input of block 78). However, it may not be possible to make all of the necessary connections if the function blocks are placed badly. Therefore, the router may have to run many times with different placements before a complete circuit is produced. The effort of routing is further complicated by the inability to hierarchically decompose the problem: there may be millions of blocks that must be connected together. For the routing tool, two multiprocessing decomposition methods were implemented. One method uses multiple processors to route a single wire, with each processor doing the routing in a different way. Because some routes may encounter dense areas or get trapped in unusual geometries, routing can be slow. By having multiple processors compete in the routing task, the fastest route will be used and the slower ones (sometimes 10 times slower) can be abandoned without waiting for them to finish. The second method of using multiple processors assigns a different route to each processor, provided that they do not intersect on the chip. By setting up a "protection frame" around each route, multiple non-intersecting routes can be run in parallel. These two methods have been combined, and together they have shown a speedup of 4x on an 8-processor system.

Conclusion

The Electric VLSI Design System has been translated from C into Java. The Java implementation has an immutable database that is split into a client and server. This database offers speed, security, and flexibility. New tools are being rewritten to take advantage of multiple processors in modern computers. The new system is open-source (like the old system) and is being used all over the world to design integrated-circuit chips.

Acknowledgments

The author wishes to thank the Java development team, which rebuilt Electric from the ground up. The full-time members include the author as well as Dmitry Nadezhin, Gilda Garretón and Russell Kao. The part-time members include Jonathan Gainsley, Michael Wessler and Ivan Sutherland. We also gratefully acknowledge Sun Microsystems, our employer, who supports the development of Electric and continues to offer it to the world, free of charge.

References

Gray, J. and A. Reuter, "Transaction Processing: Concepts and Techniques," Morgan Kaufman Publishers, 1993.

Guttman, Antonin, "R-Trees: A Dynamic Index Structure for Spatial Searching," *ACM SIGMOD*, 14:2, 47-57, June 1984.

Moore, Gordon, "Cramming more components onto integrated circuits," *Electronics*, 38-8, April 19, 1965.

Rubin, Steven, "An Integrated Aid For Top-Down Electrical Design," *Proceedings ICCAD - 83*, IEEE Computer Society Order No 518, 111-112, September 1983.