7-1-1998

Evolving hardware with genetic algorithms

Kevin Kerr

Evolving Hardware with Genetic Algorithms

by

Kevin E. Kerr

A Thesis submitted in
Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE
in Computer Engineering

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
July, 1998

Approved by:

Dr. Muhammad E. Shaaban, Assistant Professor

Dr. Peter G. Anderson, Professor of Computer Science

Greg P. Semeraro, Adjunct Assistant Professor

Abstract

Genetic techniques are applied to the problem of electronic circuit design, with an emphasis on VLSI circuits. The goal is to have a tool which has the performance and flexibility to attack a wide range of problems. A genetic algorithm is used to design a circuit specified by the desired input/output characteristics. A software system is implemented to synthesize and optimize circuits using an asynchronous parallel genetic algorithm. The software is designed with object-oriented constructs in order to maintain scalability and provide for future enhancements. The system is executed on a heterogeneous network of workstations ranging from Sun Sparc Ultras to HP multiprocessors. Testing of this software is done with examples of both digital and analog CMOS VLSI circuits. Performance is measured in both the quality of the solutions and in the time it took to evolve them.

Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface, a standard set of services provided by a software library for a particular application.
CMOS	Complementary Metal Oxide Semiconductor, the most common technology for implementing digital and analog VLSI circuits.
genome	The genetic material of an organism.
FET	Field Effect Transistor
GALib	Matthew Wall's C++ genetic algorithm library.
IAD	Interaction Diagram, a tool used in object oriented design which clearly establishes the roles and responsibilities of the objects in the system. It provides specific details on how the objects share information and pass messages to one another.
IC	Integrated Circuit
MOS	Metal Oxide Semiconductor
On–Line Performance	A measure of how much computing effort it took to find a solution.
Off–Line Performance	A measure of the final quality of the solution.
pandemic	A property of the GA such that all individuals have the opportunity to mate with any other individual. A global selection operator.

phenotype	Organism formed by the instantiation of a chromosome (or genotype).
PVM	Parallel Virtual Machine
SPICE	Simulation Program with Integrated Circuit Emphasis, a ubiquitous simulation program for analog circuits. Initially developed by U.C. Berkeley in the early 1970's.
VLSI	Very Large Scale Integration

Chapter 1

Introduction

The primary goals in this thesis are to create a system in which a genetic algorithm can be employed to design and optimize electronic circuits.

The focus will be on VLSI implementations because of the unique set of constraints which they offer [WE93], [GM93]. For instance, in VLSI, active devices like transistors are far cheaper to construct than passive components like capacitors, resistors, or even sometimes wires. As a result, whenever possible, transistors are used in place of passive devices. Often it is not obvious how to go about designing a circuit while keeping in mind all the constraints, which may include: speed, area, power, and cost. A genetic algorithm can be employed to explore the possibilities and come up with some solutions which try to meet all the requirements.

The circuits under investigation will primarily be composed of NMOS and PMOS field effect transistors (MOSFET), as would be found in a CMOS IC manufacturing process. The researched designs include simple logic gates and an amplifier. The GA software has the capacity to evolve new circuit topologies and to optimize existing circuits according to the user specifications.

1.1 Why use evolution in the design of hardware?

There are several reasons why evolutionary techniques may be considered in the design of electronic circuits:

- To explore a wider solution space than would be considered with conventional design methodology.
- The design requirements do not lend themselves to a conventional solution.
- Changes in technology may render old design methodologies obsolete, requiring new methodologies to be created.
- To automate tedious and time consuming tasks.

Sometimes an evolutionary technique may be inappropriate in design. This is the case when a known, nearly optimal solution is available. The time required to set up and run a genetic algorithm to solve the problem would likely far exceed the time required for conventional design.

For testing purposes, standard circuits will be designed and optimized with a genetic algorithm. This gives a good basis of comparison when scrutinizing the genetically evolved circuits.

1.2 Document Overview

This thesis is broken into a number of chapters. Each chapter is designed to explain one aspect of the thesis in detail. Chapter 2 gives a brief introduction into the mechanics of genetic algorithms. An example is provided to aid in the explanation. A section on schema theory, the basis for most of the theoretical work with genetic algorithms, is included. Chapter 3 focuses on other aspects of genetic algorithms. A major section includes prior

work on using GAs in engineering design. This includes work on electrical and mechanical systems.

Since the focus of this thesis is on VLSI circuits, chapter 6 is devoted to discussing important issues in VLSI design. This includes an introduction to P and NMOS FET transistors, CMOS design fundamentals. A CMOS amplifier is presented along with the relevant performance metrics which are used to evaluate such circuits. There is also a section on proper circuit simulation and testing.

Chapter 4 discusses different parallel genetic algorithms. A section is devoted to each of the major classifications, and to why one particular architecture was selected over the others.

Chapter 5 delves into the software design of the parallel genetic algorithm. An object oriented analysis of the software model is given. This includes a review of the major objects in the system and how they interact. A series of interaction diagrams (IADs) is given for each the major events in the software system.

Finally the results are given in chapter 7. The performance of the software is reported in terms of execution speed and quality of the solution. The software is run with different levels of parallelism to compute speedups and find inefficiencies. The transistor sizing problem is run for a number of different circuits. This chapter is followed by a conclusion to summarize the findings of this thesis and outline future enhancements which could be done.

Additional information is included in an appendix. This includes the source code to the software and the modifications made to SPICE.

Chapter 2

Introduction to Genetic Algorithms

This chapter introduces the fundamentals of genetic algorithms (GAs), how they work, and some of the terminology used. Genetic algorithms were first formalized by John Holland in his classic text *Adaptation in Natural and Artificial Systems* [Hol75] in 1975. Here the basic connections between natural selection, adaptation, and evolution are put in terms of searching for an optimum. In natural systems each organism represents an instantiation of its genetic material. Through natural selection, organisms which have characteristics well suited to the environment will prosper. Genetic variations occur during reproduction of organisms. As a result the children of such organisms are not identical to their parents. These variations keep the population diverse. This diversity allows the population to adapt to changing conditions in the environment. In this way, mother nature can be said to be searching for the optimum set of characteristics for a given organism in the environment.

Searching is a major component to many modern problems. For instance, the entire field of artificial intelligence consists primarily of two things: knowledge (information) representation and search. This can also said of natural systems. The representation of information is contained in the genome. Evolution supplies the search mechanism. The search problem for natural systems is compounded by the fact that the environment itself consists of other adapting organisms. It is a moving target which can be quite chaotic. Yet the evidence

for the success of natural selection is evident in the broad range of strange creatures which live on our planet. Attempts to simulate this search mechanism are put into the category of genetic algorithms. We can only simulate on a crude level, but still have found that the technique is very useful on a wide range of practical problems. Nature has many advantages over our simulations: truly massive parallelism, compact information representation (genes), and lots of time. Also, to this day, we do not fully understand the mechanism of evolution in natural systems and therefore cannot completely simulate it.

2.1 The Basic Genetic Algorithm Technique

In order to solve a problem with a GA it must first be converted into a format which the GA can work with. That format must completely describe all the configurable portions of the system into one compact data structure. The most common is just a simple binary string. This string is roughly the equivalent to a chromosome in nature.

Initially a population of strings needs to be created. Often it is just a random set of strings. Each string in this population is instantiated into the system it represents. In nature this is referred to as a phenotype, but in GAs it is usually just a point in the solution space of the problem under investigation. This solution can then be evaluated for fitness. Fitness is a measure of how good the solution (phenotype) is relative to the rest of the population. Once the fitness of each individual in the population is known, selection takes place. This is the process of picking individuals out of the population for reproduction. The selection process is designed to favor individuals with high measures of fitness while not excluding all individuals with low fitness. This is done to ensure that potentially useful genetic material is not lost by suppressing under achieving individuals.

The genetic operators of crossover and mutation can be applied to the set of selected strings. Crossover consists of exchanging sections of two different strings. Mutation refers to randomly modifying portions of the string. These operators are designed to simulate

the reproduction process in nature. Parents will exchange genetic material, and in the process errors may occur. These errors are important to introduce new genetic material into the population and prevent the stagnation of portions of the string. A new generation of strings has now been created from the old population and the process can be repeated. The population is said to have converged when every individual is almost identical to every other. The algorithm converges (finds a solutions) when the average fitness has met some predetermined or maximum level.

This is the process upon which almost all genetic algorithms work. There are many variations and extensions which have been researched. For instance there are different ways encode the chromosome, a variety of selection algorithms, and different crossover operators [Gol89a]. Sometimes problem specific optimizations are put into the GA to help speed convergence [HK94].

2.2 An Example

An example will illustrate how the mechanism works. This one is taken from Goldberg [Gol89a] and goes through all the major steps. The problem in this example is a simple one: find the maximum value of $f(x)$ in the range of integers from 0 to 31 where $f(x) = x^2$. The answer to this is trivial, but it still demonstrates how a GA would solve this problem.

The first step is to encode the input, x, into a format which can represent any possible solution to the problem. A binary string of length 5 is a good way to encode the input. Any combination of ones and zeros will produce a valid input, with complete coverage.

Now we need a starting population. Selecting a small population size of $n = 4$ will keep the example simple. Four bit strings of length 5 are then randomly generated. Each member of the population is then evaluated by computing x^2. This gives a fitness value to each individual in the population. Table 2.2 lists the initial population and the fitness value for each.

#	String	Fitness	% Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100

Table 2.1: Initial Population

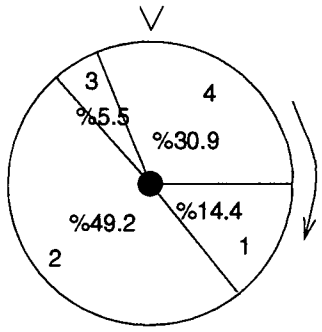


Figure 2.1: Roulette wheel analogy of the selection process

The next step is to select members of the population to be used when creating the next generation. It is desirable for the individuals with the highest fitness to be used to reproduce. At the same time, it is not good to throw away too much genetic material at once because there may be useful parts to individuals who do not score well. The probability of an individual being chosen for reproduction is made proportional to its performance. The analogy is a roulette wheel divided up into n sections. The size of each section is proportional to its relative performance to the rest of the population. The wheel is spun to select an individual.

The wheel is spun once for each individual to be placed into the mating pool. After the pool has been filled, individuals are mated at random and this is where crossover occurs. A point somewhere along the length of the gene is selected at random. Everything past this point is swapped between the mating pairs. For example, if the first and second individuals in 2.2 were mating, the following crossover would be possible:

$A_1 = 0110|1$

$A_2 = 1100|0$

In this example, the fourth position is used as the crossover point. The newly created individuals would look like:

$A'_1 = 01100$

$A'_2 = 11001$

A mutation operator may be applied to the new members before the new generation is complete. Typically, a very small proportion of the bits will be flipped at random, on the order of one chance out of a thousand. This is necessary to add fresh genetic material into the population as it evolves. For instance, if after the first iteration of the algorithm, no member of the population had the least significant bit set, then there would be no chance to find the maximum value of $x = 31$ for the $f(x)$. This is because crossover cannot add any material which does not exists anywhere in the population. Only mutation can do this. Similarly, excessive mutation can destroy useful genetic material.

In summary, in order sustain a population of 4, the selection routine would be repeated four times and the selected individual would be copied into the pool. The members of the gene pool would then be paired off. Crossover would be applied to each pair. Mutation is then performed, and the process would continue.

2.3 Schemata

The theoretical basis for GAs is in schemata [Gol89a]. This theory attempts to explain why GAs work as well as they do, and makes some predictions about their performance. Schemata are similarity templates. A schema is a pattern which an individual in the population may contain. All the members of the population which have the pattern would have the same schema. Genetic algorithms work by searching a population for successful schemata and combining them to create better individuals. The principle is that good building blocks

can be used to produce superior offspring. The difficulty is in finding building blocks that work together, and GAs do this very effectively.

A schema can be thought of in terms of a string with wild cards. Certain positions in the string are *don't cares*, they can take on any value in the available alphabet. Using the example of a bit string, possible values are either 1 or 0. A schema for all strings of length 6 starting with two 1's would look like: 11****, where the star is a wild card. In general, the total number of possible schema of length l with an alphabet of k is $(k+1)^l$. For a binary string of length 30, this would be $3^{30} = 206 \times 10^{12}$ *different* schema out of a possible search space of only $2^{30} = 1 \times 10^9$ individuals. Schema theory states that all the schemata in a given population are implicitly searched in parallel. The number of schemata in a population of size n with string length of l can vary from k^l (all individuals identical) to nk^l (all different). By processing n individuals the GA could potentially be searching up to nk^l schemata. This is perhaps the only known case of computational explosion working in our favor.

Important characteristics of schemata have been identified. The first is the order of the schema, the number of defined positions in the pattern. Given a schema H =011*1** then the order of H is 4, or $o(H) = 4$. The other characteristic is its defining length, the maximum distance between defined bits in the pattern. The defining length of H is $\delta(H) = 4$. The defining length is a measure of how vulnerable a schema is to being broken up by crossover. The order of a schema is a measure of how vulnerable it is to being destroyed by mutation.

The selection procedure and genetic operators determine how different schemata are propagated, created, and destroyed during the course of a run. Given a population at time $A(t)$ there may be m examples of a particular schema, H. This can be written as $m = m(H,t)$. The selection procedure selects individuals for reproduction proportionally to their fitness. Typically, a string A_i is selected with a probability of $p_i = f_i / \sum f_j$. The

number of expected schema to be selected for the next generation, $m(H, t+1)$ can be written as:

$$m(H, t+1) = m(H, t) \cdot n \cdot f(H) / \sum f_j$$

where $f(H)$ is the average fitness of schema H in the population, and n is the population size. If the average fitness of the population can be written as $\overline{f} = \sum f_j / n$ then the above equation can be rewritten as:

$$m(H, t+1) = m(H, t) \frac{f(H)}{\overline{f}}$$

This states that schemata which perform above average get an increasing number of representations in subsequent generations. This neglects the effects of crossover and mutation. Using single point crossover, the probability that a schema will be destroyed will be proportional to its defining length. This can be written as $p_d = \delta(H)/(l-1)$, where l is the length of the string. The probability that a schema will survive crossover is $p_s = 1 - p_d$. If the probability that crossover will occur is p_c then the overall probability of survival can be written as $p_s \geq 1 - p_c \frac{\delta(H)}{(l-1)}$. Adding this information to the above equation yields:

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\overline{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1}\right]$$

The next step is to add the effects of mutation. Mutation is the probability that any given bit in the string will be changed. Each bit survives with a probability of $1 - p_m$. The survival probability of a schema of order $o(H)$ is then $(1 - p_m)^{o(H)}$. For small values of p_m this is approximated by $1 - o(H) \cdot p_m$. Adding this to the above equation results in:

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\overline{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H) p_m\right]$$

This is sometimes called the fundamental theorem of genetic algorithms. There are some problems with this particular equation. One problem is that it does not address creation

of new schemas in the population, or what happens as the population starts to converge. Research is still very active on developing GA theory. The very nature of a genetic algorithm makes it a difficult thing to analyze. Analysis is complicated by the many enhancements which are used to improve performance. These include: parallel implementations, the use of non-binary genomes, and asynchronous algorithms. The genetic algorithm designed for this thesis uses all of these enhancements in order to satisfy the performance and flexibility goals. A theoretical analysis is not done here because that could be the subject of a thesis in itself.

Chapter 3

GA Research

Over the past 20 years a significant amount of work has be done in GA related research. The basic concepts behind GA are being exploited in different ways, resulting in a broadening of the field. The broader field is sometimes referred to as evolutionary computation (EC) and contains several sub-disciplines. Some of these disciplines are: genetic programming (GP), evolutionary programming (EP), evolution strategies (ES), classifier systems (CFS), and of course genetic algorithms (GA) [Enc]. Genetic programming, in particular, has shown promise in the application of circuit synthesis. John Koza, the founder of GP, has recently demonstrated excellent results in evolving electronic circuits [KBA+97]. A brief explanation of GP is given in this chapter, along with a review of some research efforts in using evolutionary techniques for engineering design.

3.1 Genetic Programming

Genetic programming is a special class of genetic algorithms with some potentially powerful constructs. A genetic program is just a sequence of operators and terminals on which a GA can operate. LISP is often used because it consists almost entirely of lists of expressions in this form, but almost any language can be used. Operators typically consist of addition, subtraction, multiplication, function calls, etc. These work on either terminals or sub-expressions. The structure of a genetic program can be drawn as a tree.

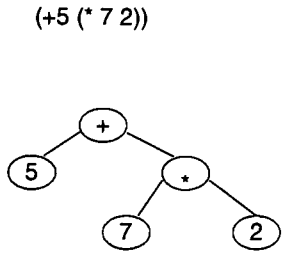


Figure 3.1: A lisp program and its tree representation

Two such genetic programs can be crossed over by swapping compatible sub-trees, thus creating children. Mutations occur by changing the value of terminals, or changing operators/function calls.

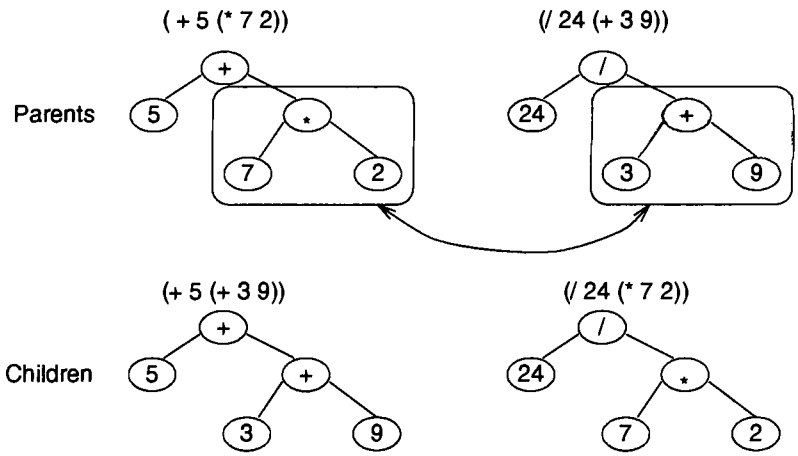


Figure 3.2: Crossing over 2 genetic programs to create children

The power of genetic programs can be greatly increased by allowing new functions to be created by the GA. This allows the possibility of generalization in the program, increasing the level of abstraction. This is important, because as the problem search space gets increasingly large, the required solution representation (gene size) increases. It has been shown that a linear increase in gene size can result in an exponential growth in the required population size [Gol89b]. The larger population sizes are necessary to ensure good initial

building blocks. By allowing fewer 'bits' in the gene to represent more abstract constructs, shorter genes can be used to find complex solutions. Human designers invariably use this concept at every opportunity in the form of hierarchy. It can be seen in engineering and computer science wherever complexity exists.

Higher levels of abstraction are often used in order to solve difficult design problems with evolutionary techniques. This sometimes requires the use of problem specific information to be used in the GA/GP. An examination of published papers dealing with circuit synthesis shows that in most cases the GA was tailored to solving the desired problem. Koza's paper is one of the best in this area as it uses a high level abstraction to synthesize a wide variety of analog circuits [KBA+97].

3.2 Past Work on Evolving Electronic Hardware

Due to the fact that electronic hardware design is not an easy task, for decades engineers have been using computers to assist in the design process. Higher levels of automation are becoming required in order to produce products with extreme levels of complexity. To compound matters, the design cycle time is being reduced as time-to-market pressure is increasing from competition. Many of the problems which designers face involve discrete optimization. These can belong to the class of problems which are the most difficult to solve: NP-complete. For instance: placing and routing conductors and components in printed circuit board and chip layout. Fortunately, many heuristic based algorithms have been developed which, while not perfect, can do a fair job in finding good solutions for these types of problems. Often, in critical situations, portions of the place and route is done by hand to ensure good results. This is time consuming and is prone to mistakes. The search for automation techniques to counter these drawbacks has lead to some interesting research. This includes the use of genetic algorithms to perform not only place and route, but also in actual circuit design.

3.2.1 VLSI Placing and Routing

A recent paper describes a genetic algorithm to solve the VLSI place and route problems simultaneously [SV96]. This is an unusual approach. Typically an engineer will floorplan a layout to give the automation tools a starting point. Major design blocks are then placed in such a way as to minimize the average distance to connected blocks. The goal is to reduce wire length as much as possible. The next step is to route the wires from block to block as necessary. Performing the place and route at the same time makes the problem more difficult, but also may lead to better solutions. The GA offers a way to do this.

The researchers recognized that it was not possible to use a simple bit string representation to encode solutions to the problem. The difficulty is that there are too many infeasible encodings of the layout. For instance, a wire is could be represented by a set of coordinates. Crossing over two valid wires could easily lead to infeasible children. The authors instead use application specific encoding and 'intelligent' operators to eliminate inadmissible individuals. The initial population is filled with a good set of building blocks to ensure that valid individuals will always exist.

A binary slicing tree is used to implement the genotype encoding. A hierarchy of sub-blocks is constructed. The root node represents the entire layout and routing. Sub-nodes contain divisions of the parent node. Routing is done at each internal node. This way the routing task is accomplished simultaneously with the placing. Leaf nodes are composed of the basic building blocks. The building blocks consist of simple functions. One of these basic blocks may have more than one layout for the same function. The genetic algorithm can select any one of the layouts depending on how it fits with adjacent blocks and where the signals are connected.

The authors used a heuristic called *iterated matching* to create the initial individuals. Blocks which have high connectivity between them are more likely to be placed together in

the tree. The connectivity was computed by summing the number of common nets between each pair of blocks. This value was then offset by a random amount. This is to ensure that each individual (tree) has a slightly different structure.

The genetic algorithm was used to layout several real-world circuits. The results were good, but the authors claimed that routing was a problem. This was because they were using a simple routing heuristic. This prevented them from making a direct comparison with commercial place and route programs. The extensive use of problem specific information in the GA is typical for highly complex problems. This is because GAs work with a representation of the solution. Some problems can be encoded very efficiently into a bit string. More complex problems may require long bit strings. As discussed above, longer strings require exponentially larger populations. For these types of problems a more abstract representation is necessary. Problem specific information is used to create a custom, compact genome which can represent most of the feasible solutions. Tree structures are popular. This approach is similar to genetic programming techniques.

3.2.2 Analog Circuit Design

John Koza is doing some excellent work on synthesizing analog circuits with genetic programming [KBA+97]. The synthesis of several important circuits was demonstrated. These circuits included: several passive filters, an amplifier, a computational circuit, a voltage reference circuit, and a temperature sensing circuit. All of these were generated using the same genetic programming software. Only the objective function and embryonic circuit were changed to create the different circuits.

One major difference between typical genetic programming and what was used for synthesizing circuits is the topology. A tree structure is used in genetic programming to create fully functional LISP programs. These programs could be described by a non-cyclic graph. Electronic circuits on the other hand consist of cyclic graphs. Instead of directly encoding

circuit elements in the genetic program, a meta-language is used to describe the construction of a circuit. An initial circuit is given as a starting point. This is referred to as an embryonic circuit. A tree of commands is constructed to operate on the embryonic circuit. A *writing head* is a pointer which indicates a modifiable element in the circuit. There are four different categories of functions in the meta-language to operate on the circuit: connection-modifying functions (CMF's), component-creating functions (CCF's), and automatically defined functions (ADF's). An ADF is similar to a subroutine. It is a macro of electronic components which can be used over and over again, and thus leveraging learned information.

Each of the different types of operators performs a different function. A CMF is used for changing the topology of the circuit. It can add new nodes to the net list at the current writing head. A CCF creates a new component at the writing head. A wire may be changed into a resistor, capacitor, or transistor, etc. A CMF changes the value of an existing component. Resistor or capacitor values can be modified in this way, but the type of component is not changed. Transistors do not have any value associated with them. An ADF attaches a whole circuit to the current writing head.

Simulation of the circuit required the use of SPICE. Each circuit had an input/output specification. Different circuits required different analysis. The temperatures sensing circuit required a DC operating point calculation at a variety of temperatures. At each temperature the output voltage is computed. The correlation of the voltage to the temperature was recorded and used as the fitness. For the filters, an AC sweep analysis was performed. The amplitude of the output was recorded for a set of points in a frequency range. Fitness was how well the output voltage matched the desired output at each frequency.

The complexity of the circuits required the use of very large populations, on the order of 600,000 individuals. As a consequence there is a significant computational effort to perform 600,000 SPICE simulations over a number of generations. A group of 64 processors was

used to perform the calculations. It still took several days to evolve some of the circuits. It took between 30 and 140 generations to evolve most of the circuits. The average number of function evaluations was 39 million.

These results are very significant. The same procedure was used to design a number of non-trivial electronics circuits, almost from scratch. The results were very good considering that no effort was put in to optimize the genetic programming parameters. Successful circuits were obtained on almost every run. However, there are several problems which probably could be rectified. No mention was made of monte-carlo simulation. Actual electronic components do not always behave exactly as specified. A 10KΩ resistor may actually be 9.9KΩ or 10.1KΩ. The same is true for all electronic components. Usually the manufacturer will put limits on the variance. These limits can be used to more accurately simulate the physical circuit. This is especially true for evolving circuits. A feasible circuit probably will not evolve in such a perfect environment. The sensitivity to any one of the variable parameters may be too high. It may be a far better task to use GA or GP in conjunction with monte-carlo simulation to evolve even more robust circuits.

A secondary problem is the range of values which the GP was permitted to use. These values were permitted to vary continuously from very small to very large numbers. In reality, discrete resistors are only available in a limited range of values. The same is true for capacitors and inductors. Additionally, some of the component values evolved with Koza's system were so large that they could never be physically built. These issues make the problem more difficult, in fact it becomes a discrete optimization problem to select the minimal number of available components while still performing the correct task.

One additional note: in VLSI, all the transistors can be custom designed. A variety of sizes can be specified depending on the needs of the circuit designer. This is in contrast to Koza's system where all the transistors were identical.

Chapter 4

Parallel Genetic Algorithm Architectures

It has long been recognized that genetic algorithms offer a great deal of parallelism. For many practical applications, the objective function evaluation is by far the dominant factor in processing time. Such applications include: aircraft design [BC89], VLSI optimization [HK94], wind turbine design [SCC96], and analog circuits [KBA+97]. Because each evaluation is independent, a number of processors can easily work together and evaluate the population with very good efficiency. The focus of this thesis on circuit design is suitable for a parallel implementation. Circuit simulation is very computationally intensive, and a serial GA would take an extraordinary amount of time given the available computers. This chapter gives some background information on existing parallel architectures. Given this information and resources at hand, a decision as to what architecture to be used can be made.

This chapter discusses the different classifications of parallel architectures. The design tradeoffs for each architecture is reviewed. A coarse grained parallel architecture is selected because it offers the best performance given the computing environment and computational requirements. Finally, some of the software packages used to implement the selected architecture are introduced.

4.1 Overview of Parallel GAs

The use of parallel genetic algorithms raises many new questions for GA designers: What type of parallel GA is best for a given application? How do I scale the number of processors to the size of the problem? How efficient is the implementation? How does the off–line performance of the parallel GA compare to a serial GA? How much speedup can be expected? A mountain of research has been done in the past 20 years to address some of these issues. Four main categories have evolved into which parallel genetic algorithms fall [CP97] . They are: global, coarse grained, fine grained, and hybrid GAs.

4.1.1 Global GAs

This is the simplest type of parallel GA. The algorithm itself is identical to a serial GA. There is a single pandemic population where any individual can mate with any other individual. A master process would be in charge of collecting the fitness from slaves. Then it would select which individuals will mate and perform the genetic operations to create a new population. The slaves would then be instructed to evaluate all the members of the new population.

This approach is good because all the research on serial GAs still applies. The algorithm itself hasn't changed at all, just the implementation. A small MIMD computer can realize good speedup if the communication overhead is minimal. However, as the architecture is scaled up, communication can be a problem. The master process must communicate with all the slaves twice each generation. Also, the master must wait until all the slaves have completed executing before selection can begin. The efficiency of this synchronous approach is hampered by objective functions which do not complete in constant time. It is also not suitable for heterogeneous environments where some machines may be faster than others.

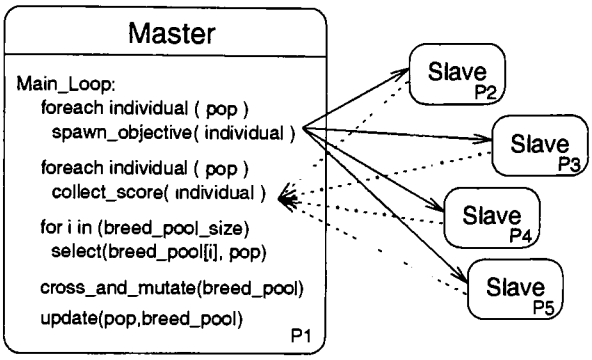


Figure 4.1: Process distribution in a global genetic algorithm

4.1.2 Coarse Grained GAs

A different approach to parallelizing GAs is to run multiple independent populations, referred to as deme populations. Periodically migration will occur where individuals from one deme may move over to another. The idea is to maximize efficiency by minimizing communication while at the same time maintaining a diverse global population.

Early work with this type of GA has shown that the frequency of migration can have a dramatic effect on the performance of the algorithm. Petty, Leuze and Grefenstette [PLG87] did work with polytypic populations. Polytypic is the biological term for a species which has a number of separated groups that sometimes inter-mate. This is equivalent to a number of deme populations with periodic migration. One major difference between this GA and the canonical serial GA is that individuals are selected on a local basis rather than a global one. Communication between deme populations in this experiment was uniform, any population could communicate with any other. This difference prevents the serial GA theory from applying. The initial research done indicates that a larger number of deme populations does significantly increase the quality of the solution at a minimal increase in execution time. Later theoretical study of this type of GA [PL89] validated the technique. It was shown that the parallel GA allocated an exponentially increasing number of trials to good performing schemata in the global population. According to DeJong, the parallel

GA which does this can be considered efficient. The experiments done by Petty and Leuze involved a high degree of communication, individuals migrated after every generation. The result of this was that the parallel GA behaved almost identically to that of a single large GA. It has not been shown what minimum level of communication is necessary to achieve this result [CP97].

Work by Cohoon and Richards [CHMR87],[CMR91] shows some promising results with a coarse grained architecture. They utilized the biological model of punctuated equilibria to show that a parallel genetic algorithm can actually produce a better result with less overall computing effort, and much less wall–clock time. Cohoon and Richards reduced the communication between deme populations such that it only occurs between epochs. An epoch occurred after each subpopulation was subjected to a predetermined number of generations. Then individuals would be copied from deme to deme. Now each deme population has extra individuals. Selection would occur within that deme to reduce the number to the set population size. A new epoch would start again. In two different papers, the authors demonstrated super-linear speedup by using a parallel GA. One was in an optimal linear arrangement problem (OLA), and the other was for a K-partition problem for VLSI layout. Later work by other researchers has repeatedly shown very good results. However, Cantú-Paz and David Goldberg contest results of such super-linear speedup [CPG96]. They claim that most such claims were a result of the researchers treating the time to find a solution and the quality of the solution as independent. As a result, most cases involving super-linear speedups had poorer results than a serial version of the GA.

The idea of punctuated equilibria is that local populations tend to converge after some period of time, and therefore the environment stagnates. The solution they converge upon may not be a good global optimum, but it is the best solution available in that population, and hence the fitness landscape is effectively flat. The individuals within the population actually compose a portion of the environment. Migration takes individuals from one stable

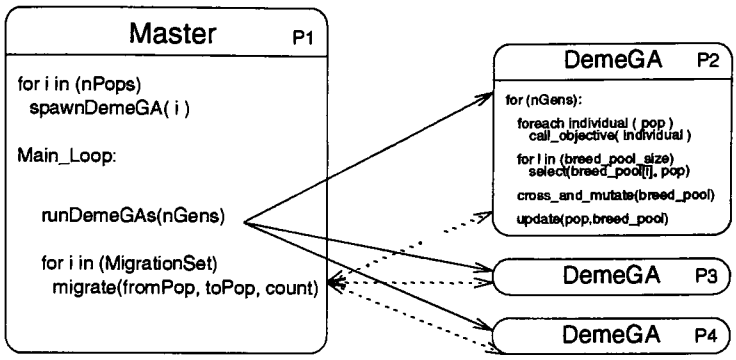


Figure 4.2: Process distribution in a coarse grained genetic algorithm

population and puts them in another. This changes the environment for the population, and stimulates genetic diversity akin to a "catastrophe" in nature. The theory originates from biology to help explain holes in the fossil record. Supposedly, a dramatic change in the environment causes rapid evolutionary change. As a result there are few examples of intermediate stages in the fossil record for many species.

A review of the literature on coarse grained genetic algorithms reveals that they can find solutions at least as well as serial GAs. In some cases, even better results can be obtained. It is far from clear exactly what configuration is best. Researchers have varied the number of demes, deme size, migration rates, migration frequency, and communication topology. The only conclusion which can be made is that good speedup is possible if the communication overhead is low and that the topology has a small diameter. Erick Cantú-Paz and David Goldberg give some bounding cases on speedups where communication delay can degrade performance [CPG96].

4.1.3 Fine Grained GAs

Fine grained GAs take the medium grain approach to an extreme. The number of individuals in each deme is very small, in some cases down to a single individual. A very high level of communication is necessary to ensure that genetic material is properly disbursed. The

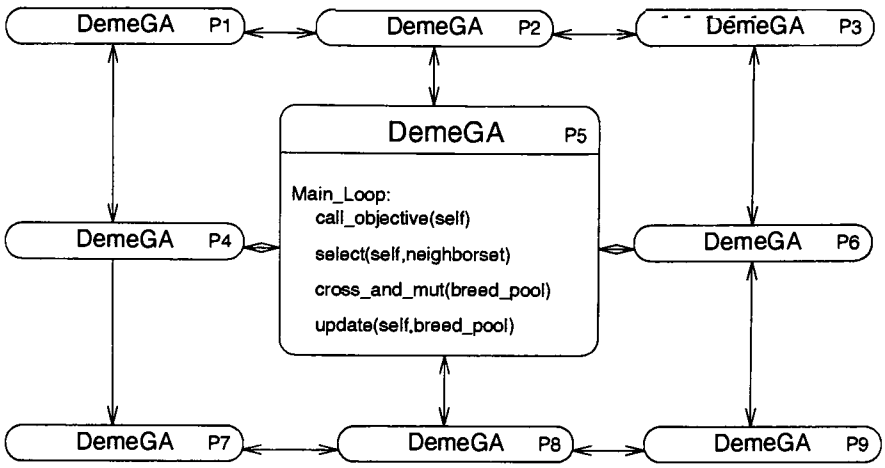


Figure 4.3: Process distribution in a fine grained genetic algorithm in a mesh topology

selection and genetic operators are also fully distributed. This type of GA is best suited to massively parallel processor (MPP) machines, where the required number of processors and a high level of connectivity is available.

Research with this type of GA has shown some good results, but comparisons between the fine and coarse grained approach have shown mixed results. In some cases, a fine grained architecture performed better than coarse grained. In other cases the opposite was true. Theoretical work has shown that the fine grained approach may have a slight advantage in finding the solution more quickly (assuming a suitable MPP exists), but it does not take into account communication costs [CP97]. The recent popularity of MPP machines may make the fine grained approach suitable for some high end applications. However, at the university level, such machines are not available. Networks of workstations are very common and are suitable for implementing a coarse grained GA.

4.1.4 Hybrid GAs

This type of GA is an attempt to get the best of both worlds. The fine grained GA is theoretically superior in finding good solutions. However, the communication cost is often

too high. A coarse grained approach give very good results with less communication. A hybrid of these two puts a coarse grained GA on top of a fine grained GA. This creates a hierarchy of populations. An example is Asparagos96 which has shown some very good results on discrete optimization problems. One problem with hybrid GAs is that they are more complex than simpler GAs. As a result, analysis and implementation are more difficult.

4.1.5 Summary

Global GAs are equivalent to serial GAs and can show some modest speedup, but they are not very efficient. Scaling global GAs to a larger number of processors will incur too much communication overhead. Coarse grained GAs have been shown to offer results as good as serial GAs. They are easier to scale to a larger number of processors and can therefore tackle larger problems. They can also be realistically implemented in a loosely coupled heterogeneous computing environment. This makes them excellent candidates for implementation in this thesis. Fine grained GAs work well for massively parallel processor machines. Such a machine is not available, so a fine grained architecture will not be considered further in this thesis.

4.2 The Selected Architecture

The selection of an architecture is constrained by several factors. These primarily include: hardware availability, communication facilities, minimum performance requirements, and software support.

The computing environment available includes workstations from a variety of manufactures. Needless to say that performance will vary considerably from machine to machine. They are connected on a 10Mbps Ethernet network. Given these computing resources it seems apparent that a coarse grained genetic algorithm with low communication overhead

will offer the best performance.

4.2.1 GA Software

A search of the Internet for GA software yielded a good C++ class library: GAlib [Wal96]. This package has very good support for most of the widely used genetic algorithms and operators. The library is designed to be portable. Most Unix platforms with a C++ compiler will have no problem compiling the library. There are rumors that it once compiled in a Microsoft environment, which could potentially add a considerable number of processing nodes. The package is extremely extensible and can be modified for user defined selection methods, crossover operators, and mutation operators. One of the included GAs supports deme populations. This is supplied by the class GADemeGA. The default behavior for this GA models stepping-stone migration. After each generation a preset number of individuals migrates from population j to population $j+1$. In this way genetic material will step from one population to the next in a unidirectional ring topology. Given n populations, it will take at least n generations for an individual to migrate to all the populations. This class does not have built in support for multiple processors. It computes all of the deme populations on the local processor one at a time. It is necessary to modify GADemeGA in order to properly distribute the task to other machines. Another software package available on the Internet makes it easy to support a large number of processors in a virtual multiprocessor computing environment: Parallel Virtual Machine (PVM) [GBD+94],[PVM].

4.2.2 PVM

PVM is a platform independent library for writing parallel processing applications. It supplies a message passing API to connect a heterogeneous set of computers together. This gives a coherent interface to write programs without worrying about communication protocols or hardware dependencies like differences in word size and byte ordering. The library is very portable and runs on almost any computer. Supported machines include:

HP 700 series, HP multiprocessor machines, Sun, SGI, and Intel. The library will also interface with different languages. These include: C, C++, Fortran, and Java. Operating systems supported include: most Unix variants, Windows 95 and Windows NT. Clearly PVM is a good way to maximize the use of computing power in an installation with a wide variety of computers. PVM is also supported on some MPP machines. This is very good if it is desirable to develop the software in an inexpensive loosely coupled computing environment and then compile it for high speed execution on an expensive supercomputer.

4.2.3 Circuit Simulation

The circuit simulation software of choice is Berkeley SPICE. SPICE is an acronym for Simulation Program with Integrated Circuit Emphasis [JQN+],[Kie94],[AH95]. The software was originally written in the early 1970's at the Electrical Engineering Department, UC Berkeley. Ron Rohrer was a faculty member who was a specialist in circuit optimization. He started a class on circuit simulation in order to develop a good automated method to optimize circuits. Students developed software over a number of years. The software which eventually came out of the project was effective, and free. As a result many people in industry started using SPICE in their own research. This gave SPICE the momentum to spur more software development. In the mid 1970's the next version of SPICE was released, SPICE2. The features included DC operating point, transient sweep analysis, and AC frequency sweep analysis. Device models include: non-linear resistors and capacitors, diodes, BJTs, JFETs, and MOSFETs. Sources included dependent and independent voltage and current. The independent sources could be set up to produce sine waves, step functions, pulses, triangle waves, and user specified piecewise linear functions. This set of features is so complete that most modern circuit simulators still accept input compatible with Berkeley SPICE2.

Development continued in the early 1980's. SPICE2 was written in Fortran so it was

decided to port it to the C programming language. SPICE3 was the result. It contains several bug fixes of some of the device models as well as more accuracy and higher speed. Some of the functionality of SPICE2 was lost in the initial translation, so most commercial vendors still claim support for SPICE2 rather than the newer version. The latest versions of SPICE3 now seems to be relatively bug free and stable. The source code is still available. Portability is very good. It uses an old dialect of C, but most compilers can deal with it. System specific code is minimal because of the batch interface. The simulator is given an input file and an output file. The input is read by the simulator and processed. The output is a text or binary file. One of the additions by SPICE3 is support for an interactive mode. The user can graphically view data plots in an X–Windows environment. This feature is optional and is decoupled from the kernel of SPICE. In this project only batch mode of SPICE is necessary.

SPICE works by reading a net list and creating a set of equations representing current or voltage at each node. A net list is a complete description of the circuit under investigation. For example a resistor is specified by giving two nodes, one for each terminal of the resistor, and a value. The value represents the size of the resistor. Capacitors are similarly specified. Transistors can have several more terminals depending on the type. Critical parameters can be specified such as: type, size, model, and other parasitics. The input to SPICE can also contain model descriptions. The descriptions supply many device physics level parameters to the internal SPICE model. Often these parameters can be extracted from a test of a physical transistor. In this way a more accurate simulation can be made of the circuit. Models can be obtained from integrated circuit manufacturers. The net list can be extracted from integrated circuit layout. This will theoretically permit accurate simulation of the complex interactions which occur in tightly packed integrated circuits.

The set of equations from the net list is converted into a matrix. SPICE performs a number of numerical techniques to solve the set of equations depending on the elements in

the circuit. These vary from Gaussian elimination to the Newton-Raphson algorithm, an iterative technique to solve non-linear equations. In some cases the algorithm will fail to converge, and a result is never found. This is probably the biggest problem many people face when using SPICE. There are a couple of techniques to eliminate non-convergence when it occurs. The primary way is to specify an initial guess of certain node voltages before SPICE begins its analysis. This allows SPICE to quickly converge on the correct answer rather than getting caught up in unusual portions of the solution space. Given the iterative nature of SPICE analysis, the execution time of any given given circuit is non-deterministic. It is for this reason that an asynchronous GA will probably be advantageous, even if all the computers running the program were identical. This way the computers do not have to be idle in order for the generations to synchronize from different machines.

4.2.4 Hardware

There are a total of five different computer architectures which can conceivably be used to run the software. These include: Hewlett Packard 700 series, HPJ282 Multiprocessor, Sun Sparc Ultra, SGI Indy 5000 and Intel Pentium. The first to reject from this list are the Intel Pentium machines. This is for two reasons. The first is that floating point performance is weak in this architecture. The second, and most important, is that they are not generally available with a UNIX operating system. This makes porting of the software much more difficult. If time permits, adding Pentiums to the computing environment will add considerable power due to the sheer number of them. If a suitable compiler is available, all of the software should be able to port with a moderate level of difficulty. The other four architectures run a UNIX variant. All of the software used in this thesis should port with minimal difficulty.

4.2.5 Synchronous vs Asynchronous

The most common form of a the coarse grained architecture is synchronous in nature. The master sends out a "step" command to all the deme populations. Each population then runs for a number of generations. The master waits for all the deme to complete the step. Now migration can occur from deme to deme to ensure a good genetic mix in the overall population. Another step command is then initiated.

- Step
- Wait for all to complete
- For each deme migrate some individuals
- Repeat

A less common implementation is asynchronous. As soon as one population completes, the master collects a migration pool and allows the population to continue. This prevents much of the unneeded idle time and greatly improves performance for some applications. Here is a rough approximation of the sequence of steps:

- Step
- Collect and buffer migration from this deme
- Send buffered migration to this deme (if any)
- Repeat

This sequence is run concurrently for each deme on the parallel machine. There is no waiting required. This drawback is that the migration must be buffered. This is easy in a PVM implementation because it will buffer messages for you. The migration can be sent as a single message. The execution dynamics are more complicated. It makes analysis of the

performance of the GA more difficult. This is probably the biggest reason it is not more common.

A timing diagram showing the differences between the synchronous and asynchronous GA is shown in figures 4.4 and 4.5, for 5 deme populations and a single master. The synchronous GA has excessive periods of idle time with simple communication patterns. The asynchronous GA wastes little time but has a more complex communication pattern. Note also that communication is better distributed in time in the asynchronous case. This improves the utilization of the communication network.

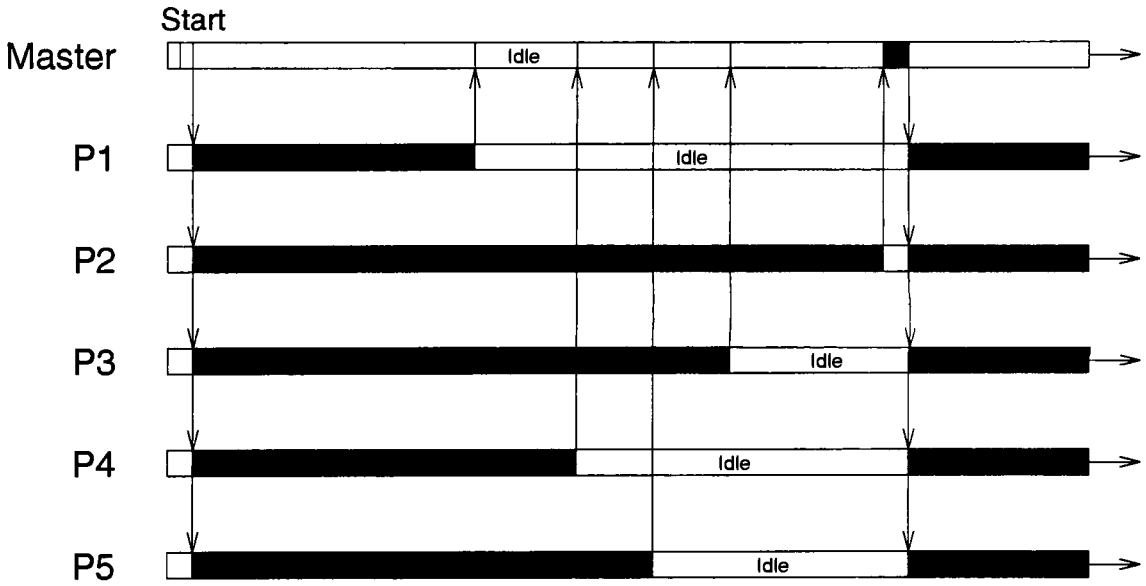


Figure 4.4: Synchronous parallel GA timing diagram

Some testing was done with both a synchronous and an asynchronous version of the parallel GA used in this thesis. For testing purposes, the genome consisted of 5 transistors with the goal of evolving a CMOS inverter. Testing was done on a number of virtual

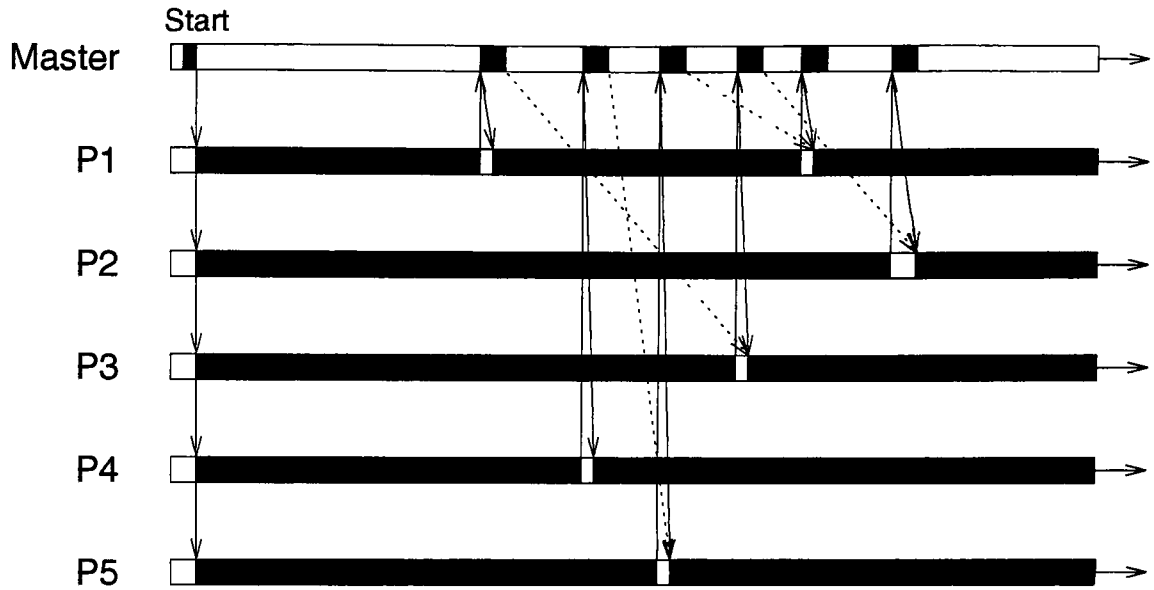


Figure 4.5: Asynchronous parallel GA timing diagram

n	T(n)	Speedup	Eff.	Redun.	Util.	Quality
1	890	1	1	1	1	0.53
2	840	1.05	0.52	1	0.52	0.30
4	1000	0.89	0.22	1	0.22	0.10
6	116	7.67	0.95	1	0.95	3.94
8	84	10.59	0.88	1	0.88	5.01
12	79	11.26	0.70	1	0.70	4.25
16	73	12.19	0.38	1	0.38	2.49

Table 4.1: Synchronous GA testing results

machine configurations ranging from 1 to 32 nodes. The units of measuring performance were in function evaluations per second. One function evaluation consisted of converting the genome into a SPICE net list, running SPICE, and computing the results.

A sample run of the two different algorithms for an 8 processor configure is included in figure 4.6. From this it is clear that the asynchronous GA offers much better performance.

Performance metrics were computed using the standard equations from [Hwa93]. These metrics are used to quantify different ways in which the parallelism is exploited for an application in a multiprocessor environment.

n	T(n)	Speedup	Eff.	Redun.	Util.	Quality
1	890	1	1	1	1	0.53
2	465	1.91	0.95	1	0.95	0.98
4	200	4.45	1.11	1	1.11	2.65
8	135	6.59	0.82	1	0.82	2.91
12	76	11.71	0.97	1	0.97	6.12
16	59	15.08	0.94	1	0.94	7.62
32	33	26.96	0.84	1	0.84	12.18

Table 4.2: Asynchronous GA testing results

Speedup:

$$S(n) = T(1)/T(n)$$

Efficiency:

$$E(n) = \frac{T(1)}{nT(n)}$$

Redundancy:

$$R(n) = \frac{O(n)}{O(1)}$$

Utilization:

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)}$$

Quality:

$$Q(n) = \frac{T^3(n)}{nT^2(n)O(n)}$$

were $T(n)$ is the time to execute on n processors, and $O(n)$ is the total number of operations performed by n processors.

This investigation shows very clearly how an asynchronous approach can greatly improve performance. This is particularly true in the application under study, which utilizes a non-deterministically long (in time) evaluation function on a network of multiprogrammed computers. The addition of other processes competing for time on these machines often amplifies the effect.

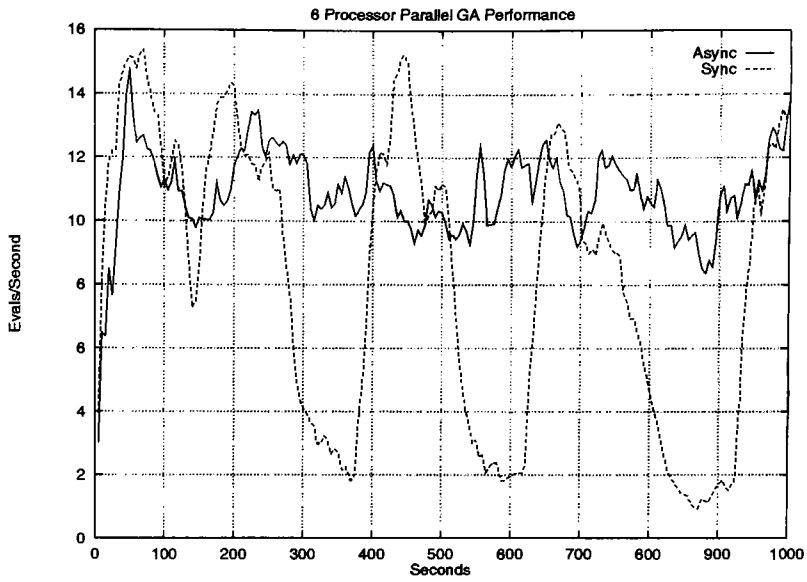


Figure 4.6: Performance of synchronous vs. asynchronous on an 8 processor system over 1000 seconds.

4.2.6 Termination

One of the more difficult problems when using GAs is the issue of termination. How do we know that the global optimum has been reached if we don't know what it is? Conventional wisdom about GAs dictates that the run is complete when the population has converged. Convergence can have several meanings. The most common is that when 95% of the individuals in the population have the same allele for each corresponding gene in the chromosome, the population is said to have converged.

Even if the population converges, there is no guarantee that the population has found the best possible solution. There exists a proof that for some problems, the canonical GA (without eliteism) will never find the global optimum, and in fact, will never converge [GR94]. How do we deal with these problems? Michael Vose has some interesting things to say on the subject of convergence [Vos94]:

> There is a "convergence" which can be observed empirically in GAs which does

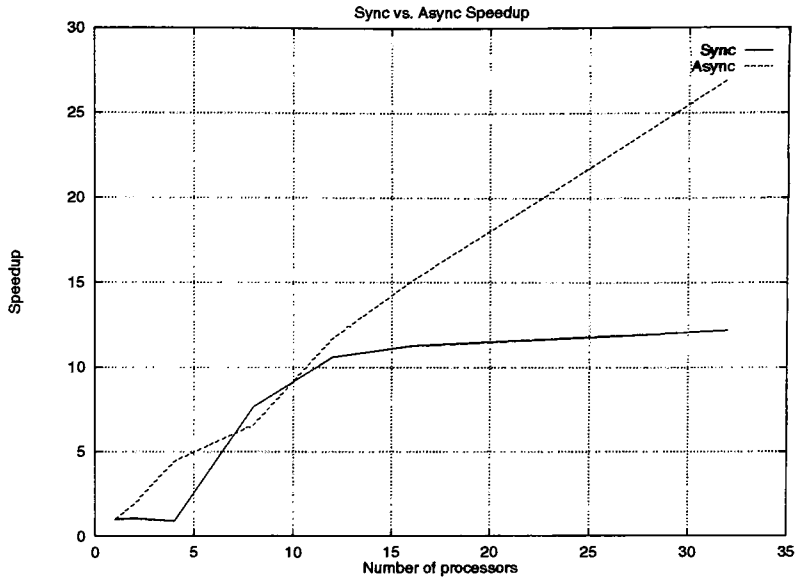


Figure 4.7: Calculated speedups for a range of system sizes.

not correspond to an unchanging population, but rather to the population's composition being "more or less the same" for an extended period of time. This is vague, and of course after a time the population will change its composition radically (if the GA is run long enough and if there is ergodicity in the chain).

I'm trying to convey the idea that a GAs behavior can be regarded as a series of "transient phases" where in each phase populations seem, more or less, to stabilize near some location in population space. An analogy for stabilization may be temperature, it never actually converges. Gentle winds cool, as the sun slips behind a whisp of white. Warmth returns, as the sun peeks out from behind a passing cloud. It is nevertheless quite reasonable to speak of temperature as being (or as having converged to) 78 Fahrenheit on a lazy spring afternoon; the temperature has stabilized near some location in temperature space.

I have done some limited simulations which suggest this temporary and perhaps vague idea of "convergence" in fact takes place. The Markov chain theory sup-

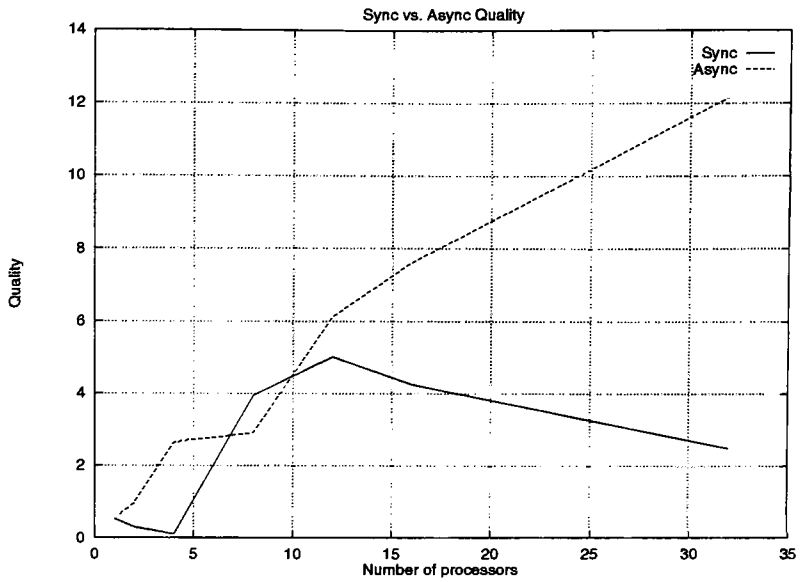


Figure 4.8: Quality factor for the two systems for a range of system sizes.

ports this phenomenon, actually predicting that it should happen. A GA should "more or less converge" for a time, and then the population should undergo a major change after awhile only to enter again into a "more or less converged" state.

What I'm wondering is, what evidence of this has the GA community observed? The answer may be "none" since conceivably when GAs are used in optimization they are always somewhere within the initial transient regime.

In other words, convergence of the population means that the search has found a local optimum. Given enough time, the GA will eventually find a better solution, assuming it exists. The question remains: how long do we wait? The conventional solution is to use a sufficiently large population such that the transient convergence is close to the global optimum. The other approach is to stop the GA when the solution is "good enough".

The problems with convergence are more difficult in distributed GAs. There is generally no global measure of convergence. For this thesis, a robust termination procedure has not

been researched. Statistical data is gathered from each deme population and maintained by the master process. For the circuits under investigation, the optimal solution is, for the most part, known. Using this knowledge, in conjunction with statistical information from the GA, the run is terminated by hand or when a performance threshold has been met. This is not a completely rigorous solution, but has been used often in the past for similar systems [KBA+97].

Chapter 5

Software Design

The needed software has several major requirements. It must be able to genetically evolve individuals in a coarse-grained parallel architecture. Individuals need to be converted into a simulation file. A simulator (SPICE) needs to be run on this simulation file and be able to return a fitness value. This value is fed into the GA and evolution of the population(s) will continue.

Three different software packages must be integrated so that they can be used as a single unit to meet the requirements. A GA library (GAlib) [Wal96], circuit simulator (SPICE3f4) [JQN+], and a message passing API to accommodate multiprocessor execution (PVM3) [PVM]. The source code to each if these packages is available from the Internet.

From the GA library a new class is created. This new class can be derived from the given DemeGA class. The supplied DemeGA class sets up a framework for using deme populations in conjunction with a central controller. The PVM library must be linked with the deme class. This gives the GA the capacity to spawn children transparently on remote processors. A child program must be written to run as a slave to the main GA. The child GA evaluates each individual in its population and communicates back to the main (master) GA. The task of taking a genome and converting it into a net list then falls to the slave GA. The slave must spawn a copy of SPICE (locally) to evaluate each individual in the local deme population. SPICE must be modified so that it can return a numerical value of

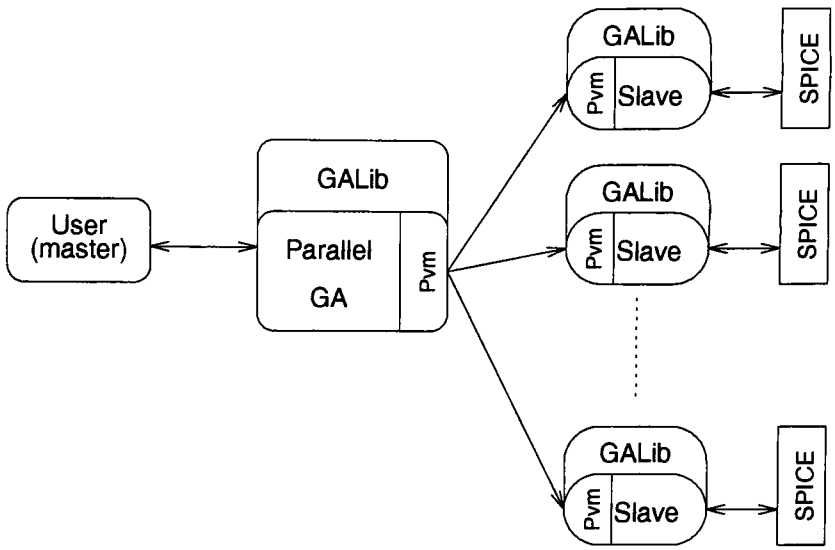


Figure 5.1: How different portions of the software fit together

fitness given a net list. This software architecture can be seen in figure 5.1

5.1 Analysis

The analysis of the software helps to put the system requirements in a concise way. Standardized notation puts these requirements in a language which anyone understand. Unified Modeling Language (UML) is a common object oriented notation which is used here.

The first step in object oriented analysis is creating use case diagrams. A perimeter is drawn around the software system. All interaction between the system and the outside world is then defined. Typically a user, or other software systems, will lie outside this line. Each major interaction with the system is recorded inside the region. A line is drawn from this interaction to the actor outside of the system who requests the service. In this case, the actor is the user and is represented by the master program. The master program creates a parallel GA object and interacts with it. Figure 5.2 shows the user "using" the system.

Each use case is described below:

- Specify GA Parameters (fig 5.7)

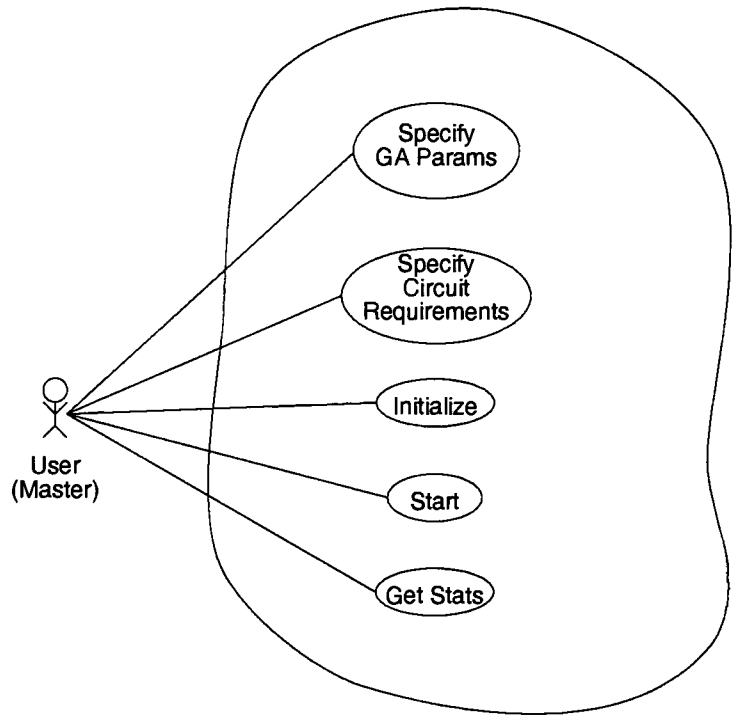


Figure 5.2: Use case diagram for software requirements

  - Mutation rate
  - Population size
  - Number of deme populations
  - Communication rate
  - Crossover method
  - Mutation method
  - Genome initialization method

- Specify Circuit Requirements (fig 5.8 and 5.9)
  - Set of Nodes
  - Genome size

  - Circuit elements (transistors, resistors, capacitors etc)
  - Input waveform
  - Desired output waveform

- Initialize System (fig 5.10)
  - Create deme populations
  - Create a master population (to keep track of the best)
  - Initialize deme populations
  - Collect initial populations from deme and place in master population

- Start Evolving (fig 5.11)
  - Send STEP command to deme
  - Record status reports from deme
  - Collect migration of best from deme
  - Record statistics of best (output best individual)
  - Command deme to migrate individuals from population to population
  - Repeat

- Get System Status (fig 5.11)
  - Output best individual and score
  - Output worst score
  - Output average score
  - Output overall performance average (Evaluations/second)
  - Output current performance (Evaluations/second)

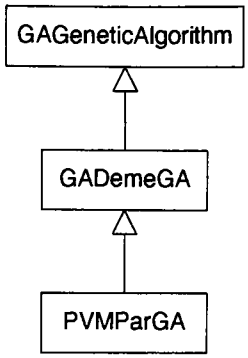


Figure 5.3: PVMParGA object inheritance hierarchy

The primary objects in the system consist of:

- Parallel GA (PVMParGA)
- Population (GAPopulation)
- Circuit Genome (CirGene)
- Circuit Elements (CirElement)
- Statistics (GAStatistics)
- Deme Populations (slave)
- Alleles (GAAllele, GAAlleleSet)

The relationship between these objects in the system is shown in figures 5.5, 5.3, and 5.6 using UML notation.

5.2 Software Objects

5.2.1 PVMParGA

The parallel genetic algorithm is derived from a DemeGA object from the GA library. This object supplies support for multiple independent populations with periodic migration.

The DemeGA does not have support for multiple processors or for elaborate migration patterns. The PVMParGA class was derived from the DemeGA class in order to supply these missing features. This new class consists of an array of GAPopulations, and another array of objects to keep track of the state of each GAPopulation. The correct solution here is to derive a PVMPopulation from GAPopulation to keep track of necessary housekeeping variables. It was decided not to do this and avoid any changes which may be necessary to the GAlibrary. A ChildStats object is instead defined to keep track of these variables. Each deme population gets a ChildStat object. All messages from children are directed to the correct ChildStat instance. The behavior of each deme is dictated by a state machine. Refer to the state diagram in figure 5.4. This is the state of the deme population from the perspective of the master process (PVMParGA). Not shown is that each individual deme may be collecting a migration from another population between the migrate and running state. The master does not really care about this state because it is transparent from the high level. The slave deme, however, must check for incoming populations before it enters the running state.

It is important to note that no other deme population comes into play at this level. Each population is completely independent. The master gives each deme a PVM task ID where it can send its migrants, and the deme takes care of it. There is no explicit synchronization between deme populations. This maximizes the performance of the parallel machine. Initial testing with a synchronous system yielded poor results. A comparative analysis of the synchronous and asynchronous implementation is given in section 4.2.5.

Migration patterns are controlled by the PVMParGA. It is currently implemented as a random migration pattern reflecting the topology of the underlying communication medium: shared Ethernet bus. Only one processor in the virtual machine can communicate on the bus at one time, but the communication cost between any two machines is uniform and short. After a deme population has completed a step, it is commanded to send a migration

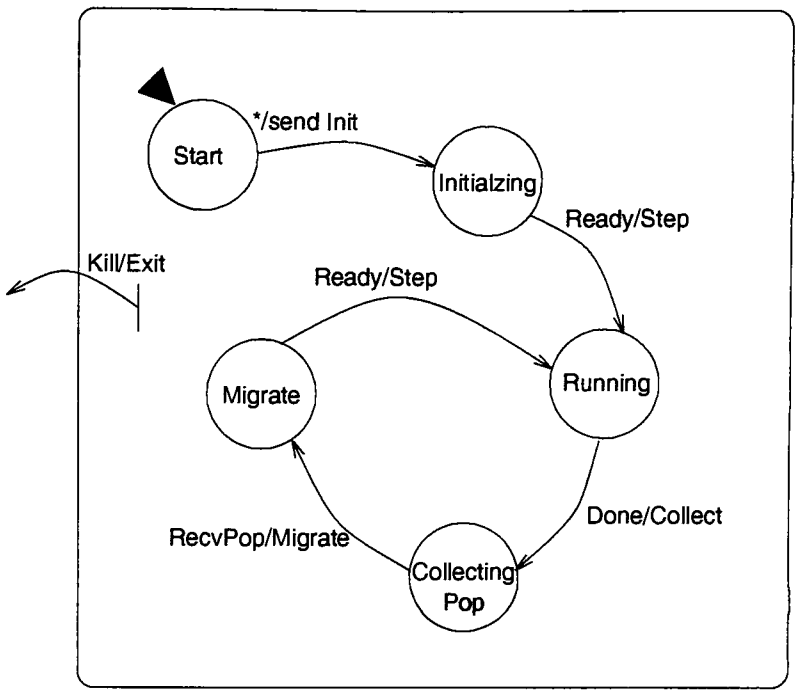


Figure 5.4: Deme population communication state machine

to another randomly selected deme. This results in a good distribution of genetic material.

5.2.2 CirGene

This is the primary genome class in the system. It is derived from GAListGenome. GAListGenome is derived in turn from GAGenome and GAList. GAGenome is a virtual base class from which all genomes come. It defines all the necessary member functions required for it to participate in the genetic algorithm. All GeneticAlgorithm based classes manipulate GAGenome objects. GAList is a template class which maintains a doubly linked list of objects. By inheriting from both GAGenome and GAList, GAListGenome exhibits the properties of a genome which consists of a list of objects. CirGene derives GAListGenome as a list of CirElements.

Each CirGene has crossover, mutation, and evaluation functions associated with it (as a pointer to a function). These functions are not member functions because they can be

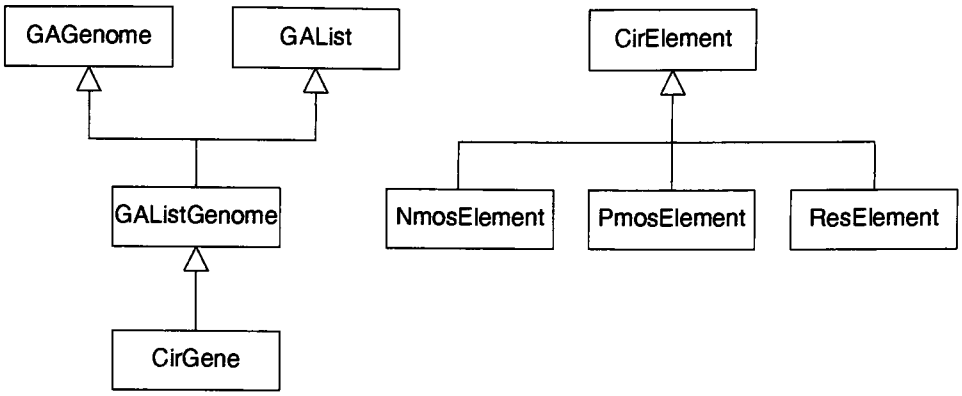


Figure 5.5: CirGene and CirElement object inheritance hierarchy

supplied by the user. This maximizes flexibility with respect to the user because a new class does not have to be derived in order to change the genetic behavior. In fact, these functions can be changed on the fly if so desired. CirGene does have static member functions defined to perform these operations. They are assigned as the default operators during construction. The default crossover performs a two point crossover. The nodes at sites of crossover are themselves crossed over. This requires CirElements to have the ability to perform crossover. The details of crossover are described section 5.4.

5.2.3 CirElement

CirElement is a virtual base class which defines the interface to a generic circuit element. An element consists of a number of connection nodes (all implemented with 3), and a number of attributes. This gene data is constrained by an allele set. There are two allele sets associated with the CirElement class. The first is a set of valid nodes. This keeps the search space down to a minimum by preventing too many different nodes from sneaking into the evolving circuit. The second is a range of values which the attribute fields can attain. Again, this keeps the circuit element attributes within a realizable spectrum of values and prevents things like 10 Farad capacitors from evolving.

The `'write'` member function produces a SPICE element line for that particular

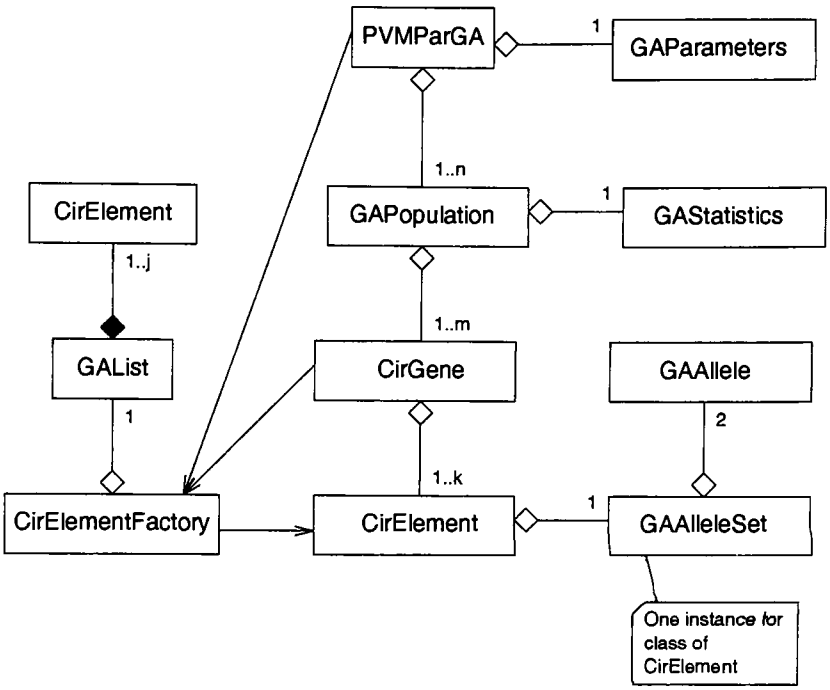


Figure 5.6: Object relationship diagram for the primary objects in the system

object. The SPICE representation of an element follows the form: `<Type>UniquieID NodeConnections Attributes ModelOptions`. For a simple resistor, the type field is the character `R` followed immediately by a unique number to identify that resistor. There are two node connections for a resistor and a single attribute field: a resistance value. All other elements follow a similar pattern.

5.2.4 CirElementFactory

A CirElementFactory is a repository of circuit elements. Each type of circuit element which is to take place in the evolution of a circuit must be registered with the factory. This is done by giving the factory one instance of each derived CirElement class. Anyone who has access to the factory can request a new instance of any registered element without having to know anything about a specific element. Each CirElement is a full fledged object which knows how to reproduce itself, mutate itself, perform crossover with any other element

and output itself. This is a powerful construct and is very extensible. The abstraction of a circuit element can be extended to include complex sub-circuits such as operational amplifiers, VCO's etc. The only requirement is that all the necessary member functions be implemented.

The relationship with the CirElement factory with the other elements is shown in figure 5.6. CirGene and PVMParGA objects can navigate to the factory so that they can request new instances of CirElements. The factory of course must know about CirElements. Each CirElement has a unique ID associated with it. The factory can create an instance of any element with the specified ID. This is useful for sending population information from one deme to another. PVM communication is described in a later section.

5.2.5 Genome Evaluation

The evaluation function for CirGene consists of converting the genome into a SPICE net list and then running a simulation. The format of a SPICE file consists of a header, a list of components, device model specification, and output commands. The member function `write( ostream os)` will write a spice file to an output stream. The header contains a title of the circuit: `Circuit Optimization`. The next lines describe the command which defines the type of analysis requested and the simulator options. The defaults are : `.tran 0.1ns 80ns` and `.options acct abstol=10n vntol=10n`. This specifies a transient analysis for 80 nanoseconds with a step size of 0.1 nanoseconds. The options line specifies the requested precision. The next items in the file are the circuit elements. Each element in the genome list is written to the output stream using the put operator '`<<`'.

Any additional, constant elements are written out after the genome elements have been written out. Such constant elements are used to simulate circuit input and output loading conditions. The input signal source and power supply would also be defined here.

The device models are output followed by the evaluation lines. The evaluation lines tell

SPICE to do a point by point comparison of a pair of vectors in memory. These vectors may correspond to node voltage, current, or even frequency domain analysis information. A typical line would look like: `.eval tran v(3) v(8)`.

Here is an example of a random circuit generated by the system:

```
Circuit Optimization
.options acct abstol=10n  vntol=10n
.tran 0.1ns 80ns
* Circuit Size: 5
M0 5 2 0 0 CMOSN l=76u w=40u
C1 2 0 690f
M2 2 5 1 5 CMOSP l=39u w=88u
R3 0 5 56
R4 2 1 58
R100 5 2 5Meg
R101 0 2 5Meg
R102 0 1 5Meg
R103 5 1 5Meg
cc 2 0 1pf
vin 1 0 dc 0 pulse(5 0 1ns 1ns 1ns 20ns 40ns)
v2 8 0 dc 0 pulse(0 5 1ns 1ns 1ns 20ns 40ns)
vccp 5 0 dc +5
.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U
+ TPG=1 VTO=0.6474 DELTA=1.6230E+00 LD=5.8150E-09 KP=8.0236E-05
+ UO=729.6 THETA=1.2540E-01 RSH=9.0910E-02 GAMMA=0.5999
+ NSUB=1.3110E+16 NFS=6.5000E+11 VMAX=2.1000E+05 ETA=9.9760E-02
+ KAPPA=1.5680E-01 CGDO=9.5924E-12 CGSO=9.5924E-12
+ CGBO=2.9552E-10 CJ=2.84E-04 MJ=0.517 CJSW=1.97E-10
+ MJSW=0.100 PB=0.99
.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U
+ TPG=-1 VTO=-0.8483 DELTA=1.8430E+00 LD=1.0280E-09 KP=1.9212E-05
+ UO=174.7 THETA=7.7780E-02 RSH=1.0500E-01 GAMMA=0.3635
+ NSUB=4.8130E+15 NFS=6.5000E+11 VMAX=4.0030E+05 ETA=1.3040E-01
+ KAPPA=9.9940E+00 CGDO=1.6958E-12 CGSO=1.6958E-12
+ CGBO=3.1527E-10 CJ=3.02E-04 MJ=0.497 CJSW=2.59E-10
+ MJSW=0.100 PB=0.99


.print tran v(2) v(8)
.eval  tran v(2) v(8)
.five  25Meg v(2) v(8)
```

The output stream is connected to a temporary file in the temp directory of the workstation. SPICE is then exec'd with the command: `spice3 -b /tmp/kek<pid>`. A unix pipe is created to connect the standard output of SPICE into a file handle within the evaluation function. The function waits on this file handle for a `Eval: <score>` line. In the event that no output has been read for an extended period of time, the SPICE process is

then killed and that genome is assigned a very bad score. Long execution times for SPICE indicate that it could not find a convergent solution for the node voltages and currents in the circuit. Nonconvergance is most commonly due to an error in the circuit specification, an invalid combination of connections and elements. Typical execution times are under a second for simple circuits with on the order of 10 elements simulating about 100 nanoseconds. Large circuits with several hundred elements, or for extended simulation times can easily take several minutes.

The output of a typical run looks like this:

```
Circuit: Circuit Optimization
Date: Sat May 30 15:14:37  1998

MAG Score: 6.3486234034645823

PHASE Score: 1763.94875624434875

Vector v(8)
EVAL: 307.71994440186750807698

Total CPU time: 0.423 seconds.
CPU time since last call: 0.423 seconds.
Current dynamic memory usage = 172032,
Dynamic memory limit = 2147282664.
0 page faults, 67 vol + 56 invol = 123 context switches.

Nominal temperature = 27
Operating temperature = 27
Total iterations = 307
Transient iterations = 300
Circuit Equations = 14
Transient timepoints = 132
Accepted timepoints = 121
Rejected timepoints = 11
Total Analysis Time = 0.11909
Transient time = 0.114194
matrix reordering time = 0.000411
L-U decomposition time = 0.015068
Matrix solve time = 0.01365
transient L-U decomp time = 0.014805
Transient solve time = 0.013339
Transient iters per point = 0
Load time = 0.041221
```

A UNIX signal handler is configured in the event that the master program sends a kill command to the deme populations. This handler will kill any running SPICE program and

delete the temporary files. This handler also catches signals from PVM in case the virtual machine is halted. With this feature the deme populations exhibit friendly behavior to all the workstations in the system.

5.3 Interaction Diagrams

An interaction diagram is drawn for each use case from the analysis section. This diagram shows the specific interaction between system objects. The roles and responsibilities of each object can be specified here. Such diagrams can be drawn at different levels of detail. In this case the idea is to convey how the system works in general, so some of the details will be left out.

5.3.1 Specifying the GA parameters

There are many parameters for the GA which can be specified by the user. These include things like mutation rates, population size, number of populations etc. A built-in object in the GA library takes care of most of this: GAParameters. It accepts a text file and command line arguments. Parameters take the form of a string-number pair. The string denotes which parameter and the number gives it a value. The GA requests the value of these parameters to configure itself. The specific interaction is shown in figure 5.7

5.3.2 Specifying Circuit Parameters

The ultimate goal of this software it to evolve circuits. The user must specify the characteristics of the circuit genome in order to evolve the desired behavior. The number of elements, the number of nodes, and the type of circuit elements need to be defined. Simulation parameters must also be specified. These include an input wave form, output waveform, simulation length and precision, and model definitions. Two different constructors are shown for the CirGene. The first, shown in in figure 5.8, generates random circuits given the number of elements. The type of permissible circuit elements is controlled by only registering the

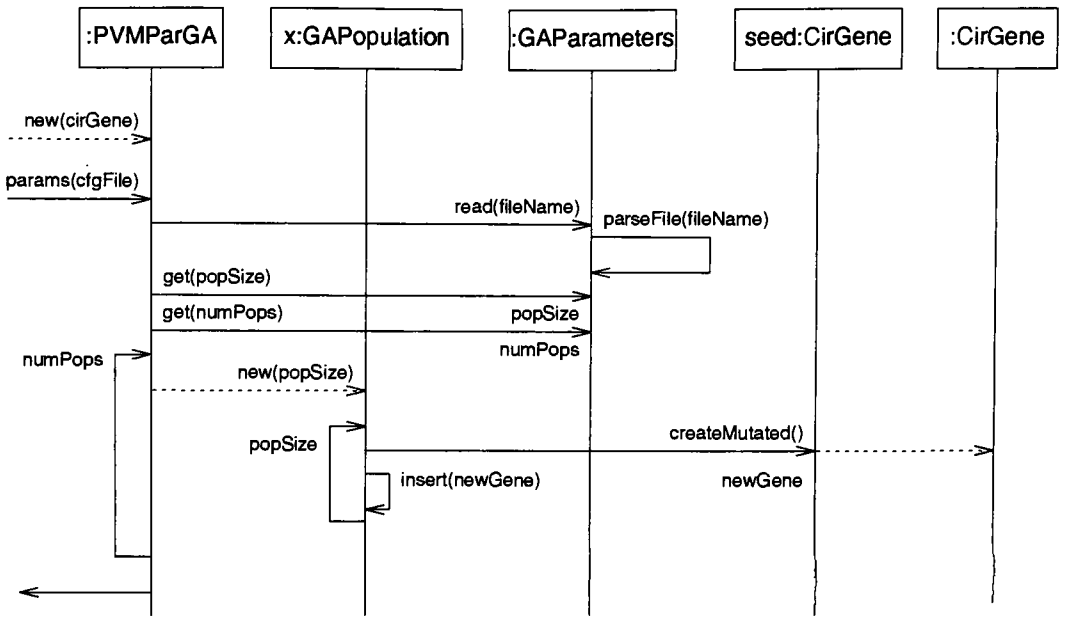


Figure 5.7: Interaction diagram for setting GA parameters

appropriate elements with the CirElementFactory.

The other CirGene interaction diagram shows how a circuit genome can be created from a SPICE file. The nodes from the spice file can be directly converted into circuit elements and added into the CirGene. Refer to figure 5.9.

5.3.3 Deme population initialization

The deme populations are started by the PVMParGA object. This is done before it is initialized, because each deme must exist before it can be initialized. The spawning of deme populations is done via PVM. PVM is referred to as a pseudo object because it is not an object oriented construct in this system, but the virtual machine it creates has many of the properties of an object. Spawning deme populations is similar to constructing new instances of deme GA's, so this notation is used. After the deme populations have been spawned, the PVMParGA informs each one of the GA parameters.

The next step is GA initialization. The PVMParGA merely instructs each deme to

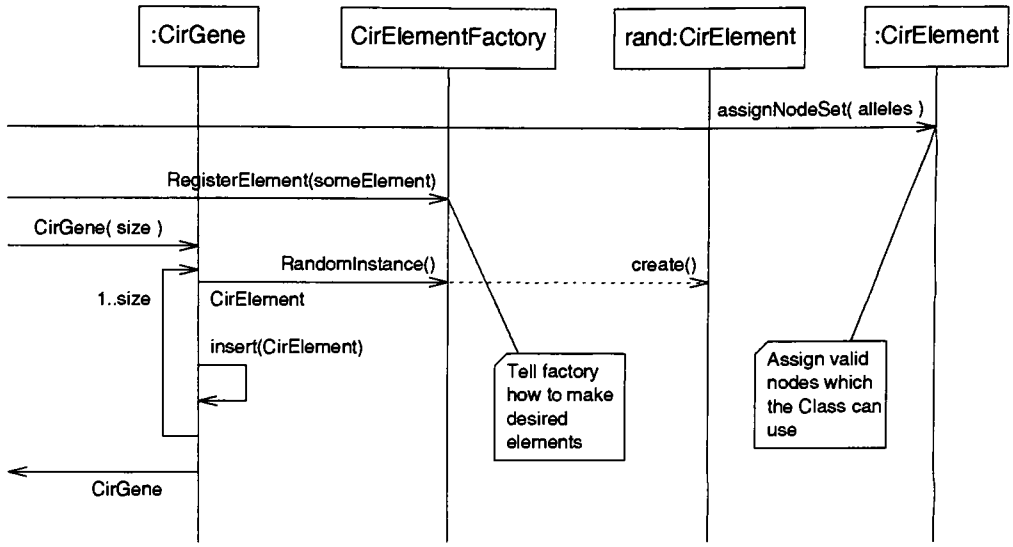


Figure 5.8: Interaction diagram of creating a random circuit genome of length *size*

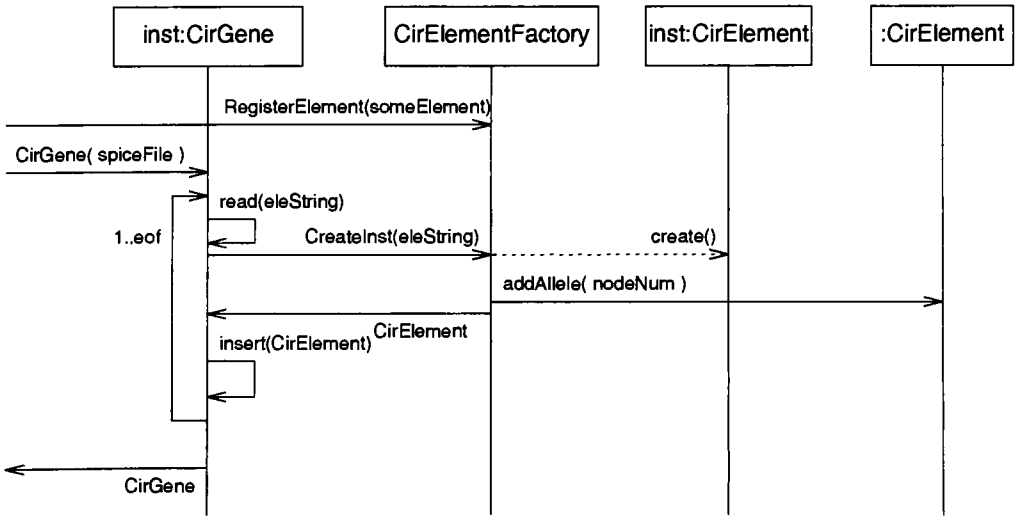


Figure 5.9: Interaction diagram of creating a circuit from a spice file

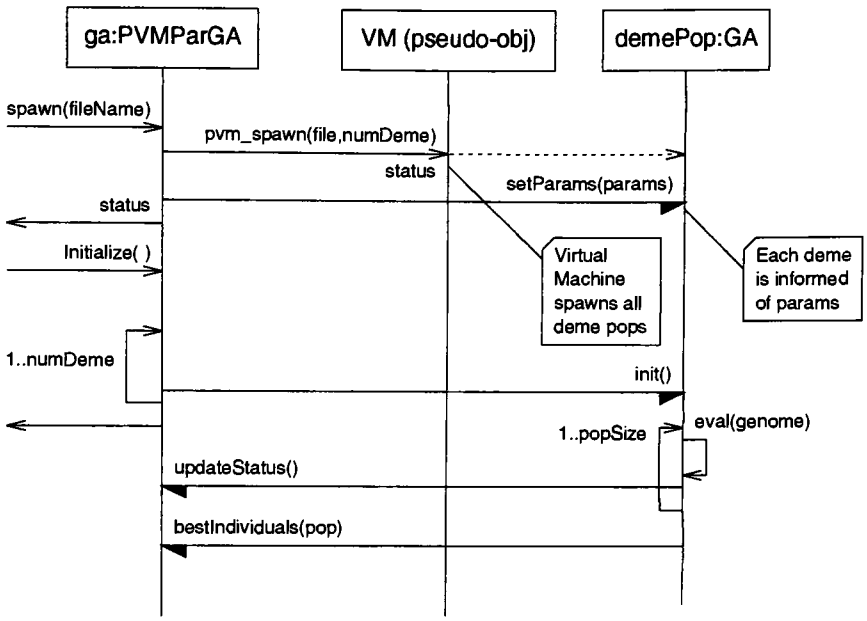


Figure 5.10: Interaction diagram of deme population initialization

initialize. The deme then run an evaluation for each individual in its population. As each deme completes an evaluation the PVMParGA is sent an EVAL_COMPLETE message. The PVMParGA object uses this information to calculate deme performance. After the evaluation phase, the PVMParGA receives the best individuals from each deme.

5.3.4 Evolution interaction diagram

In the steady state, all the deme populations will be continuously running function evaluations with short breaks for communication. Figure 5.11 shows what happens when the PVMParGA is instructed to evolve. Each deme gets a step command. A step consists of a set number of generations. The exact definition of a generation depends on the particular GA being run at the deme level. Any deme may complete its step at any time and send a message to the PVMParGA to this effect. The PVMParGA instructs the just completed deme to send a migration of individuals to another deme population.

The interaction diagram shows the asynchronous nature of the system. Periodically the

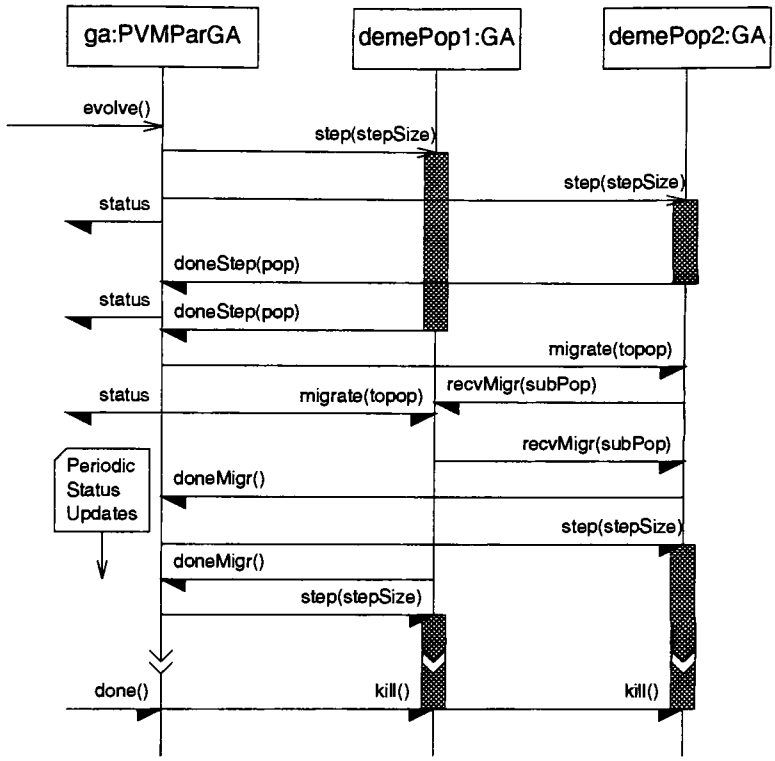


Figure 5.11: Interaction diagram of the parallel populations evolving

PVMParGA will produce status reports. These reports consist of performance numbers and of the state of the global population. Including: Best individual, worst individual, average score, overall evaluations per second, and current evaluations per second. When the fitness of the best individual becomes acceptable, the user can kill the PVMParGA which will in turn terminate all the deme populations. Alternatively, a timer can be established to run the system for a preset number of seconds after which it will terminate itself. The best overall individual is then reported in the form of a runnable SPICE net list.

5.4 Implementing Crossover and Mutation

The hierarchical nature of the CirGene object requires a hierarchical crossover. Figures 5.12 and 5.13 demonstrate this. Recall that each CirGene is actually defined from a GAListGenome. This object stores data in the form of a doubly linked list of objects. The figure

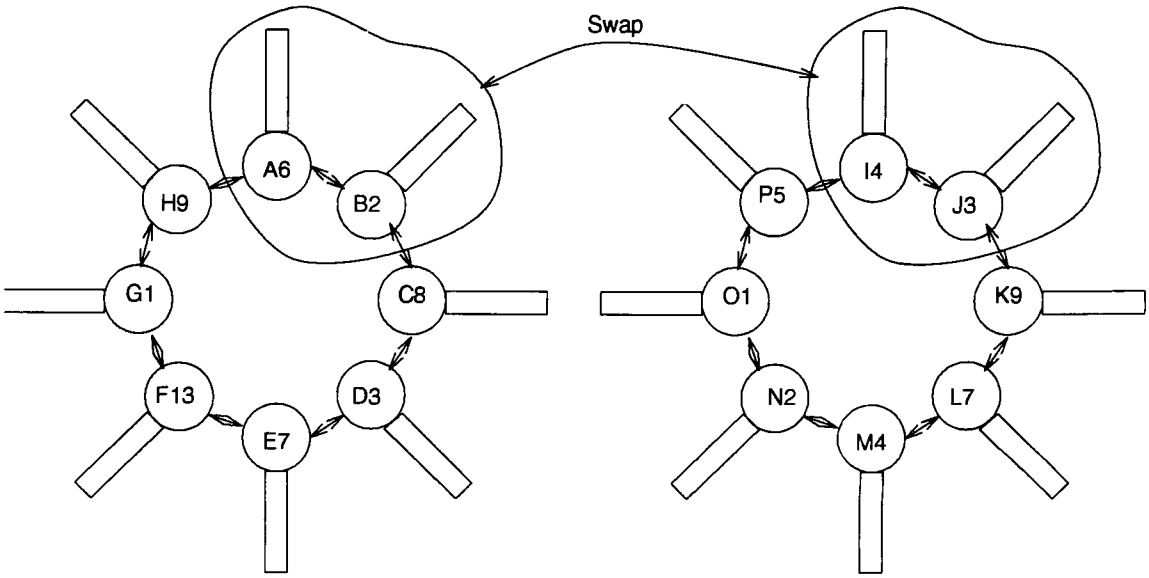


Figure 5.12: Crossing over two CirGene's at the high level

shows this as a series of circuits representing CirElements. The boxes protruding from the CirElements is the genetic material contained in each CirElement. Two crossover sites are selected in the list. The information between these two sites is exchanged between the two individuals, this is standard two point crossover. A second crossover is used to increase the resolution of crossover down to the circuit element level. The CirElement knows how to perform crossover, so the CirGene instructs two of the CirElements bordering the crossover sites to perform crossover with each other.

There are three main different types of mutation possible. The type of element in the circuit can be changed, the node connections can be changed or the attributes can be modified. The most difficult is the first because a different instance must be created to replace the old element. The CirElementFactory is used here to simplify matters. A new random element is requested from the factory. The factory selects a random instance from its list and creates a new instance given the genetic material from the element which it is to replace. This way the node connections and some of the attributes can be maintained

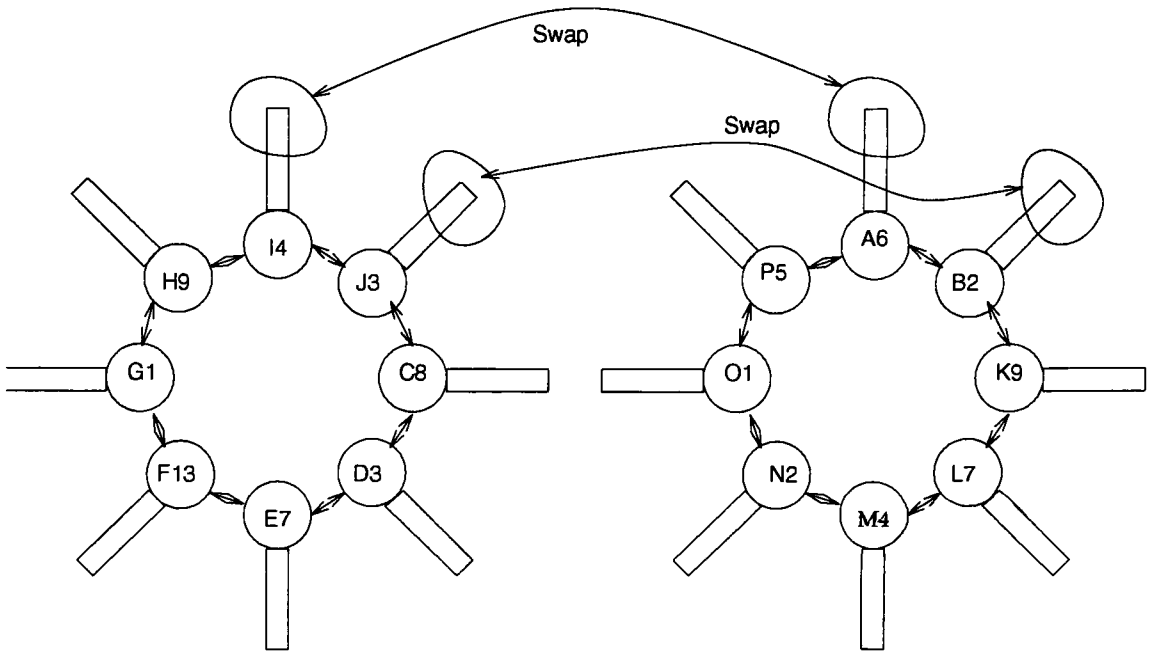


Figure 5.13: Crossing over two CirGene's at the level of the CirElements

even if the element type changes.

5.5 Circuit Characterization with SPICE

There are many issues in evaluating the quality of a circuit. A design specification may have several different and competing requirements which must be satisfied. These requirements may include such things as: input impedance, output impedance, amplitude, frequency response, and phase shift. The approach taken in this thesis is to compare the output of the circuit under test to the ideal desired output.

The evaluation of the waveform is done in two parts. A time domain comparison is followed by a frequency domain analysis.

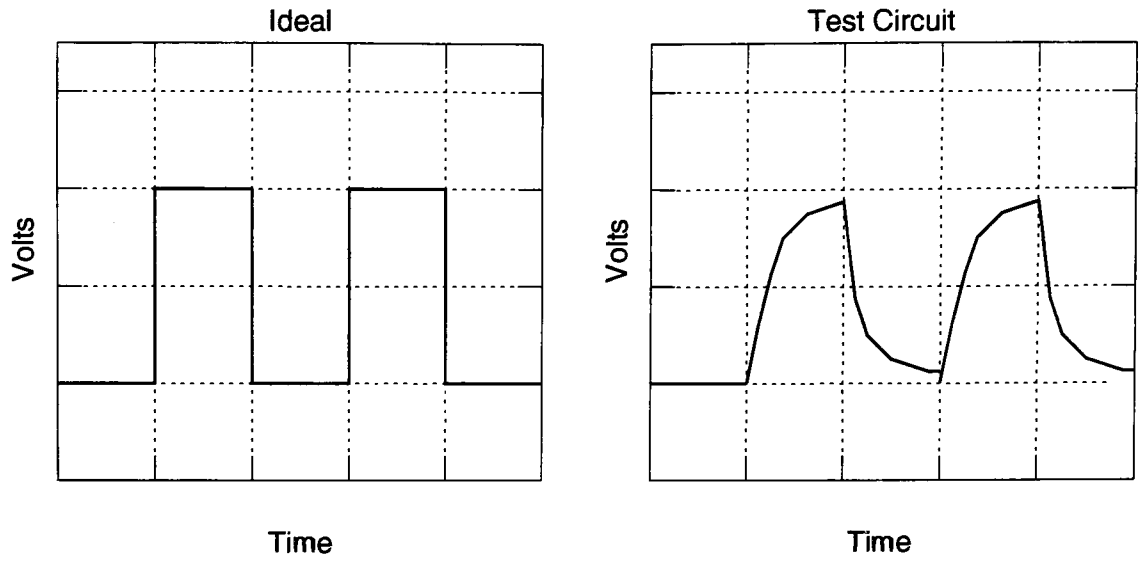


Figure 5.14: The ideal and actual circuit in the voltage time domain

5.5.1 Time Domain

For the time domain analysis, the difference between two vectors (which represent voltage, current, or amplitude inside SPICE) for the entire length of the run is computed.

$$e = \sum_{x=0}^{T} |f(x) - i(x)|$$

Where e is the error value, T is the length of the simulation vector, $f(x)$ is the output of the circuit, and $i(x)$ is the ideal output. Figure 5.14 shows the waveform of the ideal output of an inverter circuit and the possible output of a test circuit. The absolute difference between these two waves is shown as a shaded region in figure 5.15

This technique may work well for many circuits, but there are times when it may fail miserably. Figure 5.16 shows one of these cases. Circuit 1 has a poor quality signal but it is in phase with the ideal. It therefore scores fairly well. Circuit 2 demonstrates an excellent output signal. Unfortunately it is nearly 180 degrees out of phase and will score very poorly. There is no way to specify or calculate this phase relationship using such a simple analysis. A frequency domain analysis helps in this case.

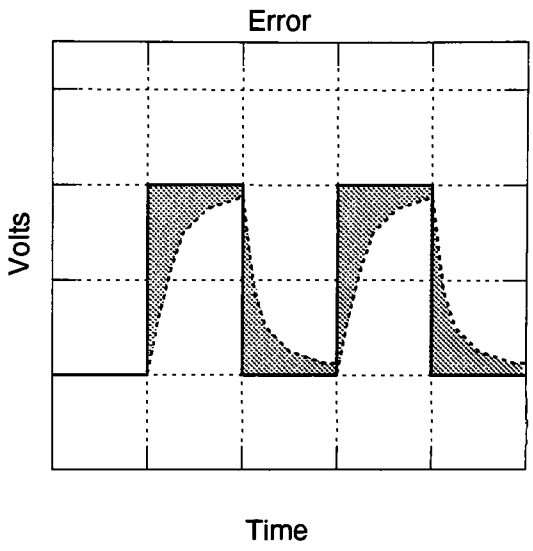


Figure 5.15: Computing the error between the ideal and actual output

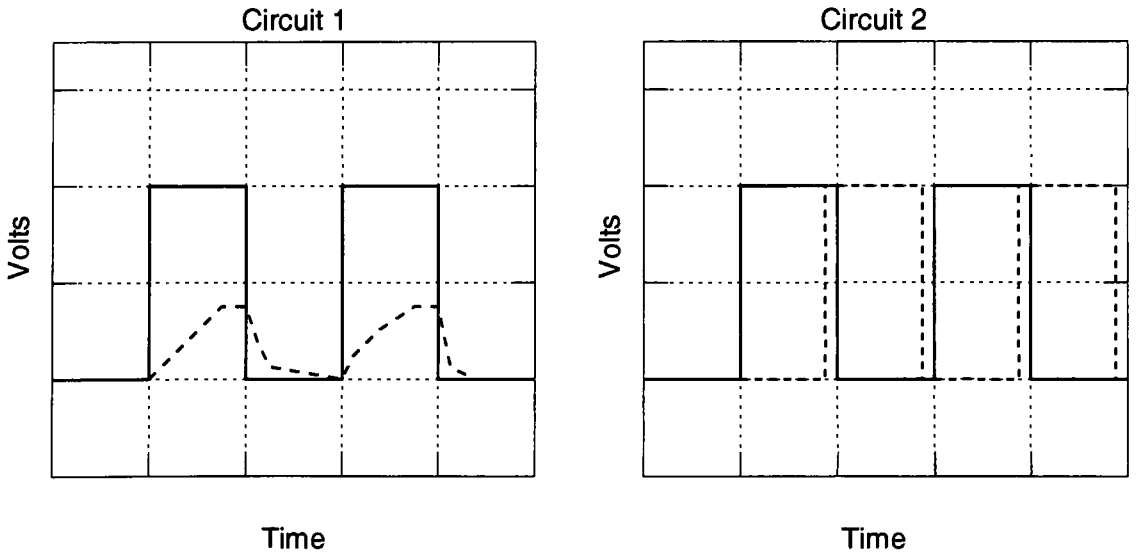


Figure 5.16: One problem with time-domain analysis

5.5.2 Frequency Domain

The waveform is converted into the frequency domain via a Fourier transform. Any time-varying periodic signal can be written as a Fourier series. One of the trigonometric forms of this series is:

$$v(t) = \sum_{n=0}^{\infty} A_n \cos(n\omega_a t + \alpha_n)$$

or

$$v(t) = A_0 + A_1 \cos(\omega_a t + \alpha_1) + A_2 \cos(\omega_a t + \alpha_2) + \cdots$$

There are two components to this series: an amplitude and a phase. The vector A_n is the amplitude of the frequency $n\omega_a$, the nth harmonic of the fundamental frequency. The phase information is given by α_n. A Fourier transform will take a signal from the time domain and generate some of the coefficients A_n and α_n. Since there may be an infinite number of such coefficients to represent the exact original waveform, only the first 10 are computed for this application. With a fundamental frequency of 2.5MHz, the harmonics will be calculated for up to 25MHz. The software can calculate an arbitrary number of harmonics, but 10 seems to work well enough.

An example of the Fourier series for a 2.5MHz square wave is shown in table 5.1. The magnitude column contains the magnitude of each harmonic in volts. The phase column has information on the phase shift of that harmonic. An example of the Fourier series of a square wave from an actual circuit is shown in table 5.2. This series is a little different because the output waveform was not precisely a square wave. This information can be used to quantify the quality of the waveform by comparing the amplitudes of the two series. The comparison is currently implemented, by computing the sum of the absolute differences for the magnitude at each point. This sum is then reported as the `MAG Score:` from the simulator. The differences of the phases are also computed and reported as the `PHASE Score:`. The phase score is not currently used, as it is fairly meaningless to compute the

sums of the differences of phase. The magnitude score is taken along with the time domain score, and this compensates for the lack of phase information.

Table 5.1: A 10-point Fourier transform data for a 2.5MHz square wave

Harmonic	Freq (MHz)	Mag	Phase Deg.	Norm. Mag	Norm. Phase
0	0	2.625	0	0	0
1	2.5	3.17029	-18	1	0
2	5	0.248032	54	0.0782365	72
3	7.5	1.02295	-54	0.322669	-36
4	10	0.242201	18	0.0763973	36
5	12.5	0.574339	-90	0.181163	-72
6	15	0.232722	-18	0.0734073	0
7	17.5	0.36997	-126	0.116699	-108
8	20	0.21994	-54	0.0693756	-36
9	22.5	0.248748	-162	0.0784625	-144

Table 5.2: A 10-point Fourier transform data for actual circuit output

Harmonic	Freq (MHz)	Mag	Phase Deg.	Norm. Mag	Norm. Phase
0	0	1.24839	0	0	0
1	2.5	1.81783	-51.89	1	0
2	5	0.639175	137.673	0.351614	189.563
3	7.5	0.466116	-87.488	0.256413	-35.598
4	10	0.309846	80.0853	0.170448	131.976
5	12.5	0.273983	-134.67	0.150719	-82.783
6	15	0.203022	26.4239	0.111684	78.3142
7	17.5	0.193292	175.767	0.106331	227.657
8	20	0.15009	-26.342	0.0825654	25.5481
9	22.5	0.147729	125.562	0.0812667	177.453

5.5.3 SPICE Modifications

Here is an overview of the enhancements made:

The syntax for comparing voltages using this in SPICE:

```
.eval tran v(<outputNode>) v(<idealNode>)
```

Similarly, current could be compared:

```
.eval tran i(<outputNode>) i(<idealNode>)
```

There is no limit to the number of evaluations which can be made in one simulation file. Each result is reported on a score line:

```
Eval: 153.2435233
```

Note: The nodes to be compared must be explicitly saved in the waveform database. I did not modify SPICE enough to have it automatically add such nodes on the fly when detecting the eval command.

For comparing waveforms in the frequency domain:

```
.five <fundFreq> v(<outputNode>) v(<idealNode>)
```

this will compute a 10 point Fourier transform with a fundamental frequency of `<fundFreq>` for each node. The sum of the absolute differences between the magnitude at each point is computed. Similarly the phase information is summed. The results are output on a score line like:

```
MAG Score: 4.8923462565283
PHASE Score: 984.76492346
```

Chapter 6

Issues in VLSI design

The technology of integrated circuits presents a radical shift in viewpoint from conventional discrete circuit design. The design constraints are almost completely different.

On a printed circuit board, manufacturing cost is computed according to the number of components and the size of the board. For a discrete design, the idea is to minimize the number of components while meeting the performance requirements. Capacitors and resistors are the most inexpensive types of components. In addition to having a low cost, they can also be manufactured to a good level of precision. Transistors are more costly. Having vast numbers of transistors on a printed circuit board is not feasible. As a result, discrete transistor designs are characterized as being biased by resistors with capacitive coupling between stages. Every electrical engineer learns this in a first course in electronics design.

This chapter describes some of the characteristics of discrete and integrated circuit design. There is review of important structures in VLSI such as buffers, logic gates, and amplifiers. Additionally, there is an overview of some of the performance criteria used to evaluate such circuits. There is also a section on the appropriate testing of circuits.

6.1 Discrete vs. Integrated Design

The two worlds of discrete and integrated circuit design are very different. Both have advantages and disadvantages. This section gives a comparison of the same circuit implemented in both discrete and integrated form.

6.1.1 Discrete Amplifier

Figure 6.1 shows an example of a discrete transistor design. It is composed of a capacitor, resistors, and bipolar junction transistors (BJTs). This happens to be a multi-stage amplifier with a differential input stage and a high gain output stage. In the first stage, R_{bias} sets up a bias current through the two input transistors. With no input signal, the current through both transistors will be equal and hence the voltage drop across R1 and R2 will be equal. The result is no output signal from the first stage into the second stage. A signal applied at input v_1 will change the current through the first transistor. Since there is a constant bias current supplied by R_{bias}, the current through the second transistor must change an opposite amount to preserve the total current. These changes in current cause the voltage below R1 and R2 to fluctuate, usually with some voltage gain from the input. This voltage is fed into a high gain stage. A capacitor is placed between these two stages to permit the proper biasing of the second stage transistor. This capacitor is a high pass filter which will allow only high frequencies through.

This circuit is typical of discrete transistor design. The circuit contains the liberal use of resistors and capacitors. It is inexpensive to build, and is appropriate for applications requiring: high current, low to medium frequency, and low to medium voltage gain. There are a number of problems. The resistors $R1$ and $R2$ should to be closely matched (nearly identical in value) in order for the input stage to function properly. Similarly, the biasing resistors for the entire circuit must have a good manufacturing tolerance in order to ensure that all the transistors are biased properly. Incorrect biasing can change the overall circuit

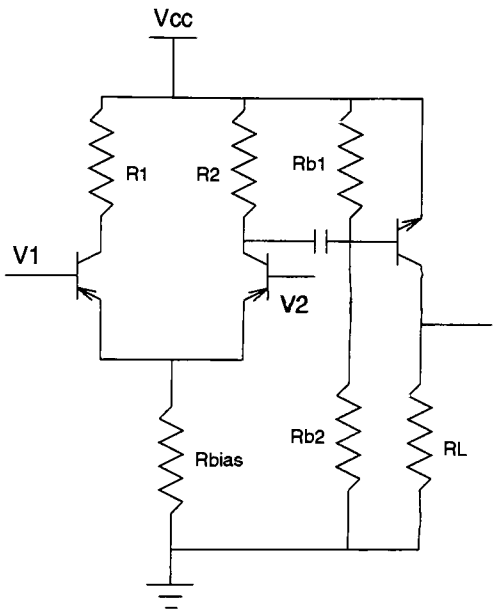


Figure 6.1: An example of a discrete transistor multi-stage amplifier

characteristics considerably. Again, the reliance on transistors for biasing makes the circuit sensitive to voltage changes on the supply rails, V_{CC} and ground.

6.1.2 CMOS VLSI amplifier

There are many differences between VLSI circuits and conventional discrete component circuits. The VLSI version of a multi-stage amplifier can be seen in figure 6.2. Notice that there are no explicit resistors. Everywhere a resistor was before, a MOSFET transistor was placed instead. This is called active loading. By placing a transistor in certain configurations, it can be treated as a special type of resistor. MOSFETS can be modeled as nearly perfect sources of current. A voltage at the gate controls how much current the FET will conduct. By fixing the gate voltage, the transistor will resist changes in current when the drain-source voltage changes, but not perfectly. The amount of current which the transistor will fluctuate is given by its output resistance. This output resistance is the effective load resistance provided by the transistor.

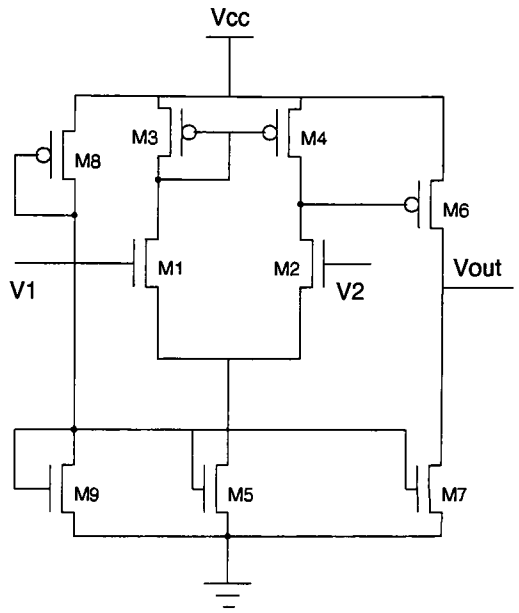


Figure 6.2: An example of a integrated circuit multi-stage amplifier

Transistors $M3$,$M4$,$M5$, and $M7$ in figure 6.2 are all configured as active loads. Transistors $M3$ and $M4$ are tied together. The result of this is that the current through both branches of the input stage is very tightly coupled. The output resistance provided by these transistors is easily in the hundreds of kilo-ohm range. As a result, the voltage gain by the input stage can be in the thousands. This is not possible in the discrete design. Load resistors of this size would destroy the proper biasing of the transistors. Realistic gains for the discrete version is less than 100. The issue with designing the MOSFET version deal mostly with selecting the correct transistor for optimal performance. Performance metrics include: frequency response, gain, phase shift, and current drive. The lack of capacitors in this design make selection of bias currents and offset voltage critical. Note that BJTs can be used in this configuration, but they are not always available in the same manufacturing process. The assumption here is that a regular CMOS process is being used and not a BiCMOS process which supports BJTs.

6.1.3 MOSFET Transistors

Transistors $M8$ and $M9$ perform a special function. They produce the basic biasing levels used throughout the circuit. The connection of these transistors turns them into resistors. The size of these resistors is determined by the physical dimensions of the transistors. The voltage at the gate of transistor $M9$ is to bias the current through active loads $M5$ and $M7$. The relationship between gate voltage and transistor current is given by:

$$I_D = \frac{\mu C_{ox}}{2}\frac{W}{L}(V_{GS} - V_t)^2$$

where I_D is the drain current, μ is the carrier mobility, C_{ox} is the oxide capacitance, W is the width of the transistor, L is the length of the transistor channel, V_{GS} is the gate-source voltage, and V_t is the threshold voltage of the transistor. From the circuit designers perspective, the only configurable variables in that equation are W, L and V_{GS}. The rest are, for the most part, determined by the circuit fabrication technology.

The physical structure of an NMOS transistor is shown in figure 6.3. The PMOS is similar. Refer to [GM93] for an analytical discussion of MOSFETs. It is comprised of a substrate, a source region, a drain region and a gate. The substrate is P-type silicone (available holes). The source and drain regions are implanted with electrons to make them N-type (available electrons). The boundary between these regions is a reverse biased P-N junction, a diode. No current normally flows from the source/drain to the substrate. The source and the drain are separated by the gate. The gate lies above the substrate with an insulator, made of silicone oxide, between them. Normally, when the transistor is off, no current can flow between the source and drain. If a voltage is applied to the gate, an electric field will be generated. This field will attract the electrons in the substrate into a *channel* underneath the gate. At a certain voltage, the threshold voltage V_t, the channel becomes temporarily N-type due to all of the attracted electrons. Now current can flow from the drain into the source. As a side effect, the gate has considerable capacitance associated

with it. The channel and the source and drain all contribute to this capacitance. This capacitance slows down the operation of the transistor.

For P-Type transistors, the process is exactly the same, but with different charge carriers, and different directions of current. The substrate is N-type, the source and drain are P-type and the channel is P-type. Holes are now the charge carriers. Recall that the voltage/current relationship in a MOSFET is related to the carrier mobility, μ. Electrons have about twice the carrier mobility of holes. So, generally, N-type transistors have considerably more current drive capacity than P-type transistors of the same size. The designer must compensate for this by doubling the size of P-type transistors when they are paired with N-type transistors.

A sample layout of a transistor is shown in figure 6.3. The layout is the lowest level of VLSI circuit design. The precise size, shape, and position of each transistor and wire in the circuit is drawn exactly the way it will be fabricated. Software tools can be run on a drawn layout such as this and calculate the parasitic resistances and capacitances which exist. These parasitics are calculated from the number of vias, the dimensions of the wires and transistors, and the details of the fabrication technology. The capacitances between drawn wires can also be calculated to compute coupling between signal lines. The net result is that an accurate simulation can be made from a drawn layout, and the designer can know what to expect when the circuit is built.

6.2 Simple CMOS Inverter

One of the most basic CMOS circuits is an inverter. It consists of a pair of transistors, one PMOS and the other NMOS. This inverter demonstrates the operating principal which CMOS logic uses. Each logic gate has two sections. A pull-up and a pull-down. Note that even though this is considered a logic gate, it is actually an analog amplifier. At the lowest level, all logic gates are actually analog devices, and can be treated as such.

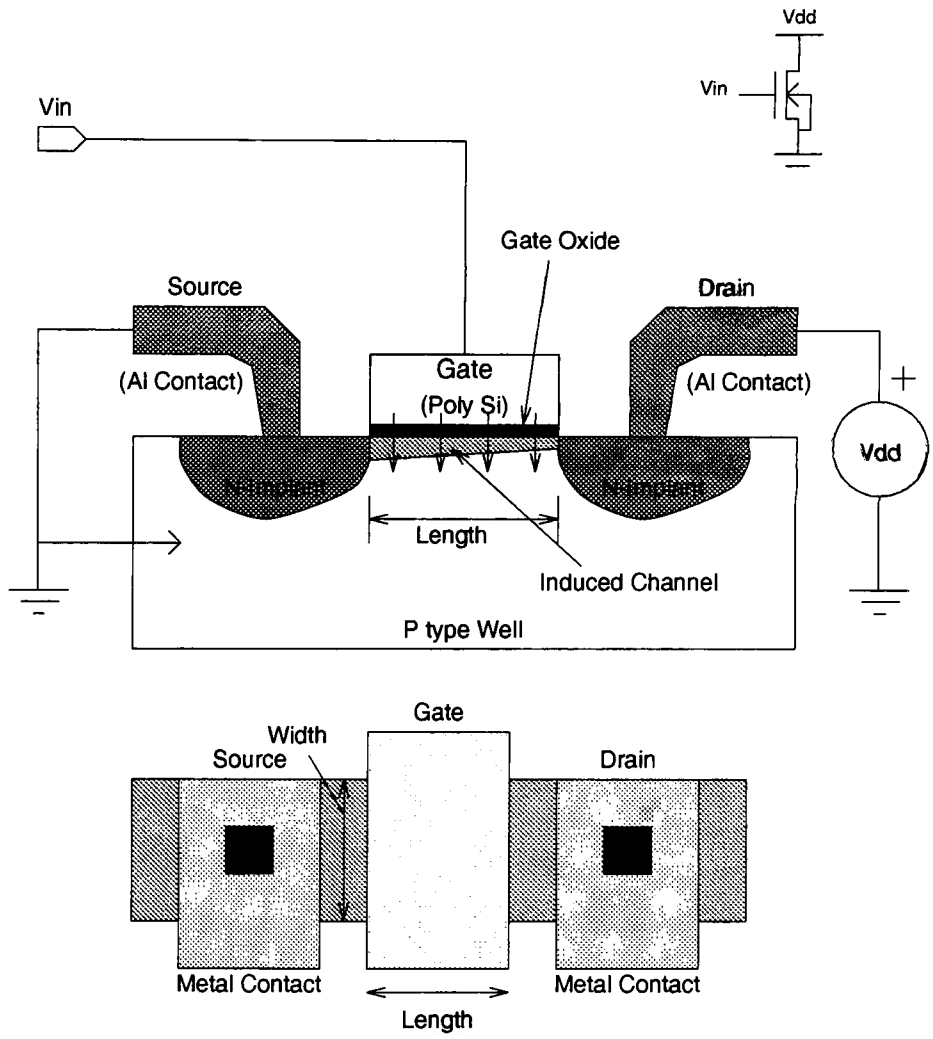


Figure 6.3: Structure of an N-Channel MOSFET, its symbol, and layout

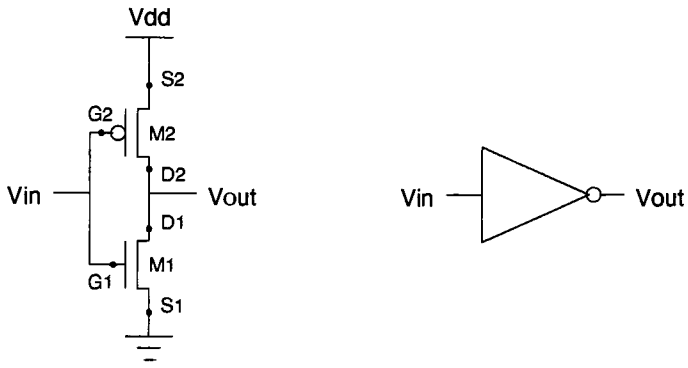


Figure 6.4: CMOS inverter

The pull-up is responsible for bringing the output voltage up to the power supply voltage. This is always done with a P-type transistor. An N-type can not be used because as the output node voltage gets within V_t volts of the supply voltage, the NMOS transistor would shut off. With an PMOS transistor, the source is connected to the power supply. This means that the voltage at the drain, which is the output voltage, will not effect the operation of the transistor. It will be able to pull the output voltage up to the power supply voltage very quickly.

The pull-down is responsible for bringing the output node voltage down to ground. Again, this is done only with NMOS transistors. When V_{GS1} is greater than V_{t1} (threshold voltage), then $M1$ will turn on and ground the output node. When V_{GS2} is greater than V_{t2}, then $M2$ will turn on, otherwise it will be off. Typical operating voltages range from 2.5 to 5 volts for CMOS logic. A value of 5 volts on V_{in} would turn on $M1$ and turn off $M2$. The result is that the output node is connected to ground. Similarly, if V_{in} is at ground, then $M1$ would be shut off and $M2$ would be on.

The change in output voltage is not instantaneous. One of the biggest factors in determining the speed of a logic gate is the load which it is driving. In CMOS, the load is almost always a capacitor. This is because CMOS gates have a significant parasitic ca-

pacitance associated with them. The amount of capacitance is related to the size of the transistor. Larger transistors have a larger gate/source/drain area and as a result have more capacitance. The gate-source capacitance is:

$$C_{gs} = \frac{2}{3} W L C_{ox}$$

where W is the channel width, L is the channel length, and C_{ox} is the unit area capacitance for the gate oxide. The size for C_{ox} depends on the thickness of the oxide layer, which is a function of the fabrication process. The designer has no control over this. The designer does have control over W and L, and must make wise decisions on exactly how to set them.

6.2.1 The Transistor Sizing Problem

The output node of the inverter is a capacitor. Transistors behave like current sources, so they can supply a constant amount of current to or from the load capacitor. The voltage on a capacitor is:

$$V = \frac{Q}{C}$$

where Q is the charge on the capacitor, and C is the size of the capacitor. Current is in units of charge per second, so the charge is the integral of current over time.

$$Q = \int_{t1}^{t2} i(t) dt$$

Since the transistors supply a constant current source, the capacitor charges or discharges linearly with a slope equal to the transistor current drive. Recall that current drive is a function of the transistor size:

$$I_D = \frac{\mu C_{ox}}{2} \frac{W}{L} (V_{GS} - V_t)^2$$

So, in order to charge the capacitor quickly, the ratio of $\frac{W}{L}$ should be maximized. On the other hand, in order to minimize the capacitance, WL needs to be minimized. Also, PMOS transistors (which have a smaller μ) need to be sized differently than NMOS transistors

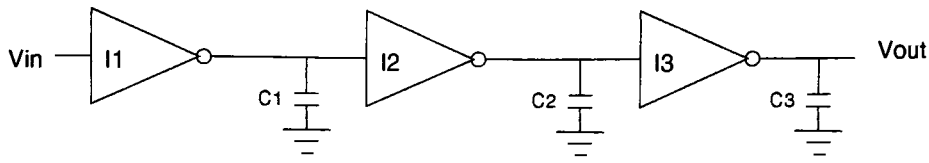


Figure 6.5: An inverter buffer chain.

in order to keep the pull-up times equal to the pull-down times. For a given fabrication technology, there is a minimum transistor length. The designer must select an appropriate transistor width in order to minimize system delay and maintain even rise and fall times for the output voltages.

If the output of a logic gate runs to many destinations, it may have an excessive capacitive load to drive. A buffer chain is often used to minimize the overall delay. Each inverter in the chain is larger than the previous. There is an optimal incremental step size. See [WE93] for a detailed explanation. A first order analysis gives an optimal step size of $e = 2.718....$ The W/L ratios of each stage should be 2.718 times larger than the last. In practice, depending on the fabrication technology, each stage should be from 2 to 5 times larger than the previous. This is one of the test problems which will be given to the GA.

6.3 Resistance and Capacitance

There are two kinds of resistance in VLSI circuits. There are intentional and parasitic resistors. parasitic are undesirable resistances which come from wires and connections. Intentional resistors are part of the design, and have a specifically designed value.

6.3.1 Parasitic resistance

These resistors come from the type of material which is being used to conduct charge. There is little the designer can do to get rid of these, other than to design a circuit topology which can be laid out with a minimum number of wires and connections.

There are a number of materials which are used in CMOS. These include: aluminum,

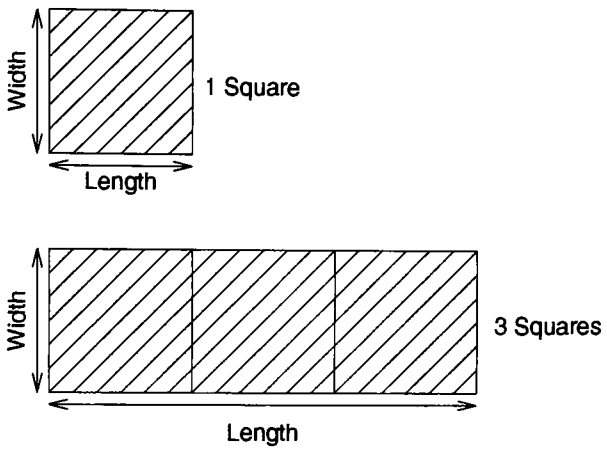


Figure 6.6: Sheet resistance.

polysilicon, the N and P active regions, and the wells. Each of these has a specific *sheet resistance* for a given fabrication process. The sheet resistance is the number of ohms the material has for a square sheet of the material, and is often put in terms of *squares*. The net resistance is calculated as the ratio of the length over width multiplied by the sheet resistance.

Long signal lines have high length/width ratios, so aluminum is used. Polysilicon is required for the transistor gates, but has a sheet resistance hundreds of times higher than aluminum. Connections between different layers have a parasitic *contact resistance*. This is usually equivalent to about one square.

6.3.2 Resistors

There are four different resistors which can be created in CMOS. These include: diffused resistors, polysilicon resistors, well resistors, and MOS resistors. All but MOS resistors exploit the same mechanism which causes parasitic resistors.

Diffused resistors are created from the same material which is used in the source and drain of MOS transistors. A long, thin structure is designed for the desired resistance value and power handling capacity. The sheet resistance for diffused resistors is in the range of 100

to 200 ohms/square. Resistors over about 1K start to get expensive in terms of consumed area. Many transistors could be place in that area. Also, the tolerance for such devices is low, about ±%20. They also have a high temperature coefficient, about 2000 ppm/degree C. Polysilicon resistors have similar characteristics. The sheet resistance is lower, about 20-80 ohms/square.

Well resistors have a much higher sheet resistance, on the order of 10K/square. The tolerance, however, is very poor. The resistor can vary by as much as %30 from the designed value. Also, there is a high voltage and temperature coefficient.

MOS devices can be used in the place of resistors in some situations. When MOS transistors are in a particular operating region they exhibit properties similar to a resistor, namely, current proportional to applied voltage. The voltage/current relationship in this mode is given as:

$$I_D = \mu C_{ox} \frac{W}{L}[(V_{GS} - V_t)V_{DS} - \frac{1}{2}V_{DS}^2]$$

MOS transistors enter this region, the triode or linear region, when the drain to source voltage is greater than $V_{GS} - V_t$. The resistance which can be obtained from a MOS device is considerably higher than the other types of resistors, while consuming a fraction of the area. The effective resistance can be changed electronically by changing the gate voltage. These types of resistors are very much non-linear, so the designer must be very careful when using this type of resistor. The GA may find interesting ways to make use of the non-linear properties.

6.3.3 Capacitors

Capacitors are frequently used in memory designs. A capacitor is used to store information in the form of electric charge. Capacitors are also used in many analog applications, including: digital to analog converters, charge pumps, amplifiers, and many more. Unfortunately, capacitors suffer many of the same problems which effect resistors. These include:

low tolerances, large area, high temperature and voltage coefficients. Capacitors are often formed by two layers of polysilicon separated by a thin layer of oxide. The gate of a MOS transistor can also be used as a capacitor. However, this type of capacitor exhibits a large voltage coefficient, and is non-linear.

With all of the problems with resistor and capacitors in VLSI design, it seems important to minimize their use. If large values of resistance or capacitance are necessary, they are often placed off-chip. One important consideration is that even though the absolute tolerance of on-chip resistors and capacitors is very poor, the relative matching can be as good as 0.5 to 1 percent. A designer can take advantage of this fact and create structures which only use relative resistance and capacitance.

6.4 Design Tradeoffs in VLSI

There are three primary resources which are considered in the design of VLSI circuits: area, power, and time (speed). Different applications will have different priorities for each of these. These resources often compete with one another. In order to increase speed, more and larger transistors can be used. This has the side effect of consuming more power and speed. Power and speed can be computed directly from SPICE simulation. Area is proportional to the number of transistors, but can not be accurately computed without an actual layout. In modern design, wires often consume far more area than transistors. Sometimes a regular structure with more transistors can be laid out in a smaller area than one with fewer transistors and a more complex topology.

It may be an interesting extension to this thesis to account for these tradeoffs. A heuristic could be used to estimate the area, and the power and speed can be obtained from SPICE simulation. The user would then rank area, power and speed in order of importance and the GA could come up with the appropriate circuit.

6.5 Simulation of VLSI circuits

Simulation is a critical step in the design process. The cost of making prototype chips can be astronomical, even though the unit cost during manufacture is extremely low. It is important to get the design right the first time.

The circuit simulator SPICE can be configured to simulate a very wide range of circuits. One of the MOS transistor models has 42 different variables which can be set. These variables can be set to match almost any fabrication process. The manufacturer can extract the values for the model settings from actual production lines. The use of extracted model data assures the accuracy of the behavior of transistors in simulation. The data from the manufacturer can also be used to calculate parasitic capacitance and resistance from the circuit layout.

Even if the simulator has been configured to exactly match the manufacturing process, care must be taken to correctly simulate the circuit. One of the most common mistakes is to neglect the input and output loading effects of a circuit under simulation. For example, if the inverter in figure 6.4 was simulated as is, it would have fantastic performance. This is because there is no load on the output node of the inverter. Placing a capacitor equal to the known load at the output would be an appropriate measure to take. If the load is unknown, then a nominal load should be used. One good nominal load is a minimally sized inverter. Such an inverter has a NMOS transistor of the minimum allowable size for the current manufacturing technology. The PMOS transistor is sized to have the same current drive as the NMOS. This gives a baseline of the minimum delay to expect due to loading.

A new simulation would result in much more realistic results, but there is still a major problem. When the simulator directly drives the input of the inverter, it does so with an ideal voltage source. This will have the effect of instantly charging the parasitic capacitance at the input of the inverter. As a result, only half of the propagation delay through the

inverter is calculated. Again, the solution is to add an input equivalent to what is expected in reality. A nominal inverter can be used if the input conditions are unknown. The signal delay is still measured from the input of the inverter to the output.

Another important consideration in simulation is to account for the process corners. Even if the device models are derived from an actual manufacturing run, the values obtained are only valid for that particular run. Different runs will result in slightly different device models. The manufacturer has a specific range in which each of the parameters can fall. These are referred to as the process corners. There may be two parts to the corners: the absolute variance and relative variance. Typically, the absolute variance in the process parameters is far higher than the relative parameters. The relative parameters represent the maximum variance which can be expected on the same chip. The absolute applies to differences from run to run.

It may be very difficult or time consuming to simulate the circuit under all of the possible conditions, including process corners, temperature, and voltage variances. There is a large number of independent variables to consider. The GA presented in this thesis could be a useful tool in finding the worst and best case scenario of all the variable. Monte-carlo simulation has been around for a long time to try and perform this type of analysis. This is equivalent to a random search. A GA may do a much better job at finding the full range of possible circuit performance.

Chapter 7

Results

This chapter contains the results from a number of tests run on the software system. There are basically two types of testing: free topology and fixed topology. Free topology testing was only performed with a CMOS inverter. Fixed topology testing is performed on a number of circuits, including: an inverter, a buffer chain, a logic gate, and a CMOS amplifier.

The results yield data on the performance of the genetic algorithm and the effectiveness of the parallelism. This data can be used to find the weakness and strengths of the system. Future work can take advantage of this knowledge by addressing problems and utilizing concepts which work.

7.1 Evolving a Free Topology Inverter

A CMOS inverter, as discussed in chapter 6, is evolved from scratch. An environment is created to elicit the desired circuit performance from the genetic algorithm. This is shown in figure 7.1. It is desirable characteristics in the inverter include: high input impedance, low output impedance, and symmetrical swing.

A voltage source, V_{src} is used as a stimulus for the circuit. It is configured to deliver a series of square pulses into the inverter. The high voltage is 5.0 volts, and the low voltage is 0 volts. The period of the wave is 2.5MHz with a 50% duty cycle (high half the time). The rise and fall times are set to take exactly 10 nanoseconds.

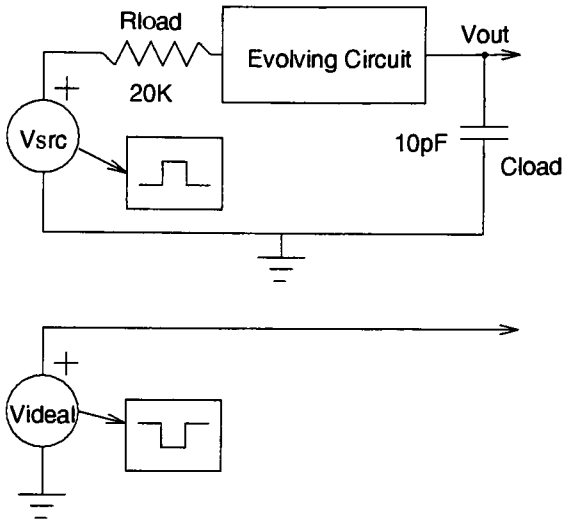


Figure 7.1: Setting up an environment to evolve an inverter.

The input resistor, R_{load}, in included to ensure that there is a high input impedance. If the input impedance of the circuit was too low, excessive current would flow across the resistor. The flow of current would cause a voltage drop to occur, and insufficient voltage would reach the input of the inverter. The net result is that the inverter would have poor output characteristics. If there is a high input impedance, then very little current would flow through R_{load}. With a low current through the resistor, the voltage at the input would remain high. With a sufficiently high input impedance, the resistor has no effect on the operation of the circuit.

A capacitor is placed on the output of the inverter. This capacitor acts as a load which the inverter must drive. This is appropriate because the loads in CMOS are, for the most part, capacitive. The load forces the GA into evolving an inverter with a low output impedance.

Another voltage source, V_{ideal}, gives the GA a reference point. This source represents the ideal output of the circuit. In this case, it is a 5.0 volt, 2.5MHz, 50% duty cycle square wave which is 180 degrees out of phase with the input signal. In other words, it is identical

to the input, except it is inverted.

The input format for this environment comes in the form of a SPICE net list:

```
Circuit Optimization
.options acct abstol=10n  vntol=10n
.tran 1ns 800ns
* 5                Tell the GA to play with 5 transistors here
r100 5 2 5Meg    * These resistors are used to set the initial conditions
r102 5 1 5Meg    * The output is weakly set to 5 volts
r104 3 1 20K     * Rload
cc 2 0 10pf      * Cload

* Here is the input voltage source
vin 3 0 dc 0 pulse(5 0 10ns 10ns 10ns 200ns 400ns)

* here is the ideal voltage source
v2 8 0 dc 0 pulse(0 5 10ns 10ns 10ns 200ns 400ns)

* This is the power supply, 5 volts
vccp 5 0 dc +5

* This are the device models from an actual
* manufactures specification
.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=1
+ VTO=0.6474 DELTA=1.6230E+00 LD=5.8150E-09 KP=8.0236E-05
+ UO=729.6 THETA=1.2540E-01 RSH=9.0910E-02 GAMMA=0.5999
+ NSUB=1.3110E+16 NFS=6.5000E+11 VMAX=2.1000E+05 ETA=9.9760E-02
+ KAPPA=1.5680E-01 CGDO=9.5924E-12 CGSO=9.5924E-12
+ CGBO=2.9552E-10 CJ=2.84E-04 MJ=0.517 CJSW=1.97E-10
+ MJSW=0.100 PB=0.99

.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=-1
+ VTO=-0.8483 DELTA=1.8430E+00 LD=1.0280E-09 KP=1.9212E-05
+ UO=174.7 THETA=7.7780E-02 RSH=1.0500E-01 GAMMA=0.3635
+ NSUB=4.8130E+15 NFS=6.5000E+11 VMAX=4.0030E+05 ETA=1.3040E-01
+ KAPPA=9.9940E+00 CGDO=1.6958E-12 CGSO=1.6958E-12
+ CGBO=3.1527E-10 CJ=3.02E-04 MJ=0.497 CJSW=2.59E-10
+ MJSW=0.100 PB=0.99

* request a transient analysis of the circuit
.print tran v(2) v(8)

* do a Fourier transform on output and ideal nodes, and report results
* fund. freq=2.5Mhz
.five 2.5Meg v(2) v(8)

* do a time-domain analysis of the output and ideal waves
.eval  tran v(2) v(8)
```

This net list is given to the GA for optimization. There are no transistors in this net list, the first comment after the header indicates that the GA should create 5 random circuit

elements with which to evolve. The remainder of the net list is copied into memory, and distributed to all of the deme populations.

The GA was configured with 32 processing nodes. One deme is assigned to each node. Each deme has a population size of 100. The mutation rate was set to be 2%. The crossover rate was 100%. The GA used in the deme populations was a steady state GA, with a 10% population turnaround/generation. After 10 generations, each deme would randomly send up to 50% of the best from its population to another randomly selected deme.

Figure 7.2 shows one of the best circuits after the initial 10 generations. The graph next to it contains two waveforms. The solid line is the ideal output. The dotted line is the actual simulated output of the circuit. The GA at this point is searching for individuals which can affect the output voltage. Most random circuits will have a nearly constant output voltage because there is no circuit structure to propagate the input signal. If, by chance, one of the transistors is connected to the output, and the input, it will score better than the average individual. Any change in the output will give it a better score because of the frequency-domain analysis. If the output is nearly constant, the Fourier transform will produce coefficients with zero value. If the output does change, then the coefficients will have a value.

The circuit in figure 7.2 scores above average because it contains frequencies that match the ideal, even though the circuit is not acting as an inverter. When the signal is supposed to be high, it is falling. When it is supposed to be low, the output is rising. What is important here is that the GA found a circuit which modifies the output. Portions of this circuit may be used as good building blocks for a more successful circuit in future generations.

The output of the simulator for this circuit is:

```
MAG Score: 5.65925170016891687652
PHASE Score: 663.21353355064184142975
EVAL: 224.55810967596593741291
```

The product of the **EVAL** and the **MAX** scores is taken as the circuit fitness. The computed

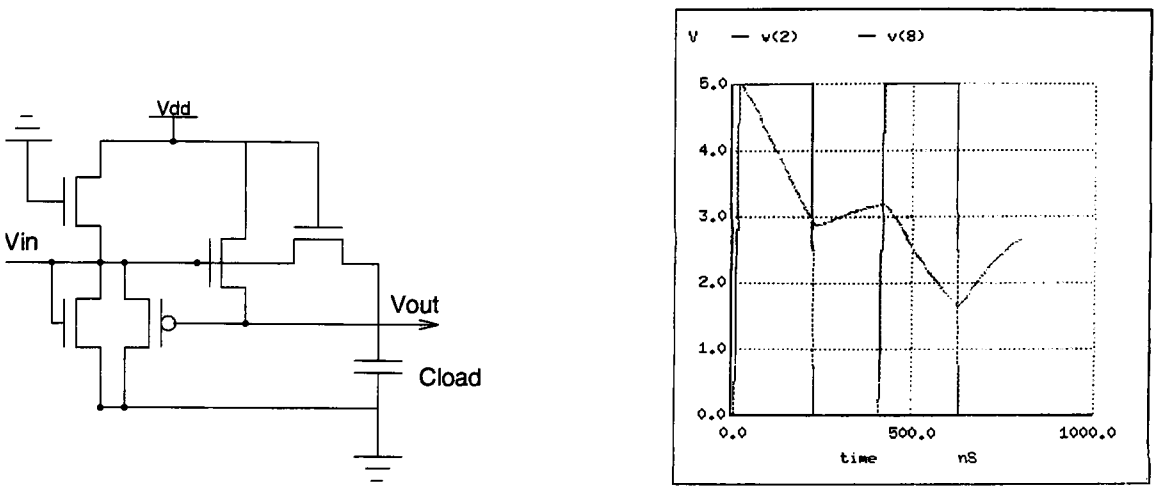


Figure 7.2: One of the heroes after the first generation.

fitness is then: 1270.83.

Significant improvements can be seen several generations later. Figure 7.3 shows one of the next successful circuits. This circuit actually behaves like an inverter. Transistor $M1$ is an NMOS pull-down transistor. The most important devices in this circuit are transistors $M1$ and $M2$. The gate of this component is tied directly to V_{DD}, so it is always on. It continuously pulls the output node down to ground. Fighting it is transistor $M2$. This is a PMOS transistor in a pull-up configuration. The gate is connected to the input. When the gate voltage is high, $M2$ is turned off. When the input is low. $M2$ is on and fighting transistor $M1$. In this case, $M2$ is a stronger transistor, with a greater channel width, so it wins this battle and is able to slowly charge the load capacitor. The voltage is also changing faster than the initial hero, which gives it a better score from the Fourier analysis.

It is interesting to note that this circuit is similar to the inverters used in a technology that does not offer both types of transistors. It is an older fabrication technology which has a much simpler manufacturing process because there is only one type of transistor. Two transistors are used, one is configured as a resistor to constantly pull the output node in one direction. The other is connected to the input signal and a power line. Just as in figure

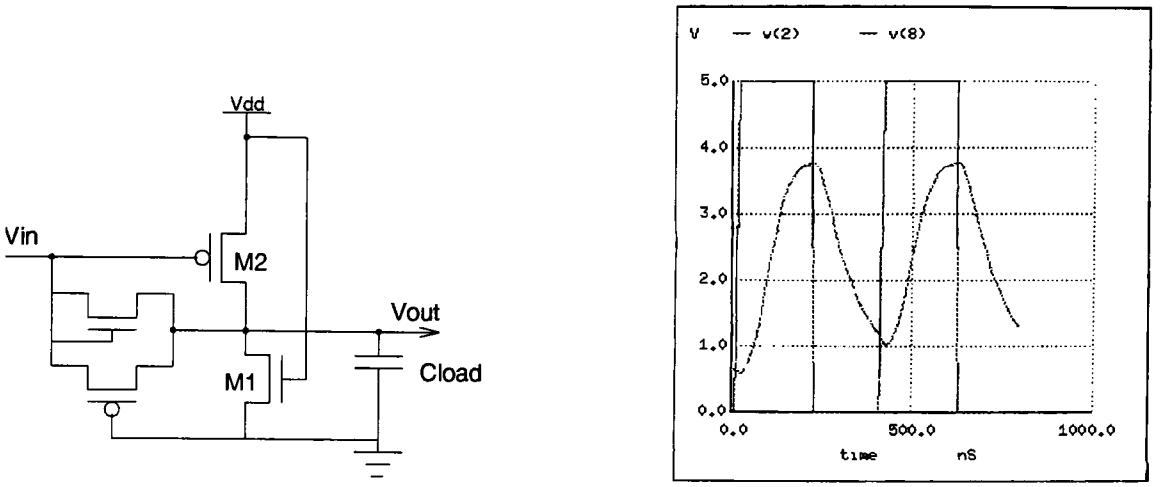


Figure 7.3: Continuing evolution of the inverter.

7.3, when the input was a one level, the resistor would dominate. At the other input level, the transistor would win.

The output from the simulator for this circuit is:

```
MAG Score: 4.86068752491190281262
PHASE Score: 1161.18570160513081646059
EVAL: 210.76642168067951388366
```

The computed fitness is: 1024.47.

A significant individual is shown in figure 7.4. This is the first occurrence of the complete topology of a CMOS inverter. The output voltage now makes a full swing from rail to rail, resulting in a much better EVAL score than before. There are extra transistors in this circuit which play almost no role in shaping the waveform, they are basically just added capacitance. It is just a matter of time until these transistors are evolved out.

The output from the simulator for this circuit is:

```
MAG Score: 2.22821010857266132987
PHASE Score: 849.63478579150387304253
EVAL: 207.05855345092655284134
```

The computed fitness is: 461.37

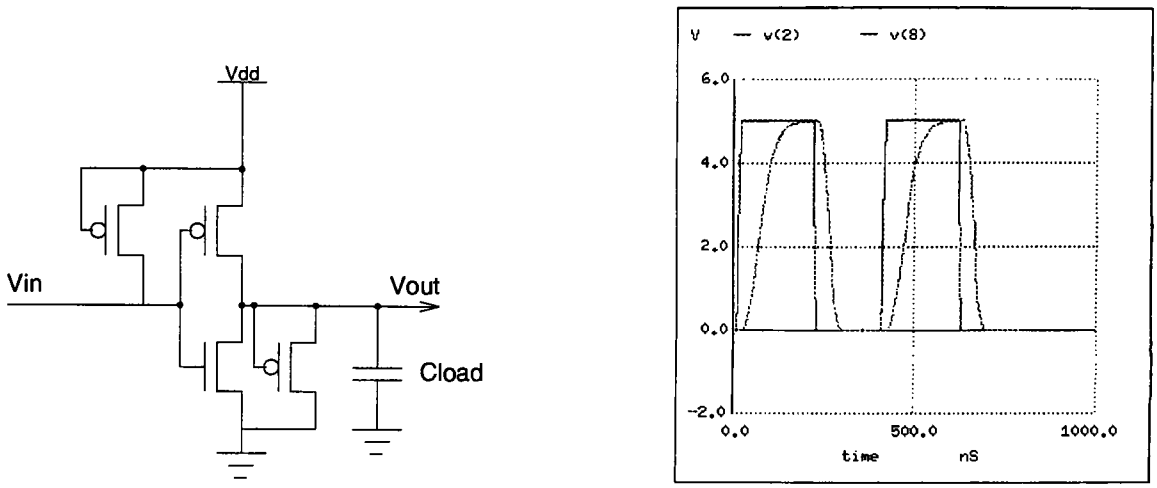


Figure 7.4: The complete inverter topology has evolved. Stray transistors remain which help rise and fall times. Transistors not sized properly.

Continuing the evolution, the circuit in figure 7.5 shows much improvement. The basic structure from the last circuit is maintained. The sizes of the transistors is the primary parameter which is changing at this point. The GA is discovering that short channel lengths and large widths produce much sharper edges. In this circuit, a stray transistor is attached. This transistor is likely a remnant from an older hero which used the resistor/transistor configuration discussed earlier. In this situation, it helps the fall times of the inverter and makes it score slightly better in the frequency domain. It hurts the rise time, so the time-domain is being impacted slightly.

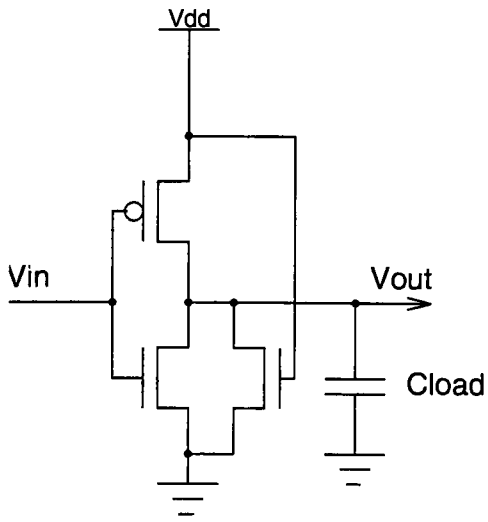
The output from the simulator for this circuit is:

```
MAG Score: 0.58893235466298421432
PHASE Score: 1216.93157178131468754145
EVAL: 206.07543072893250268862
```

The computed fitness is: 121.364

The output from the simulator for this circuit is:

```
MAG Score: 0.13652122280223158435
PHASE Score: 875.35231401584883315081
EVAL: 160.53804909737104367196
```



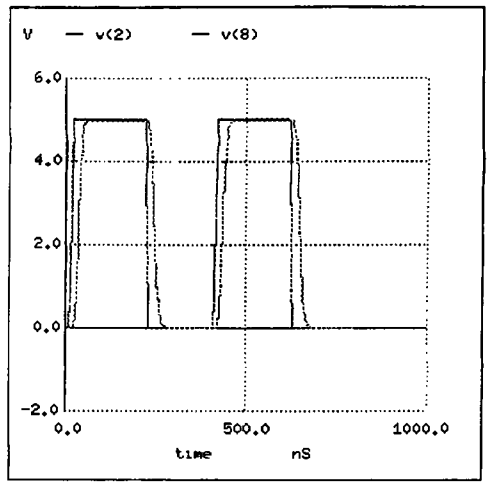
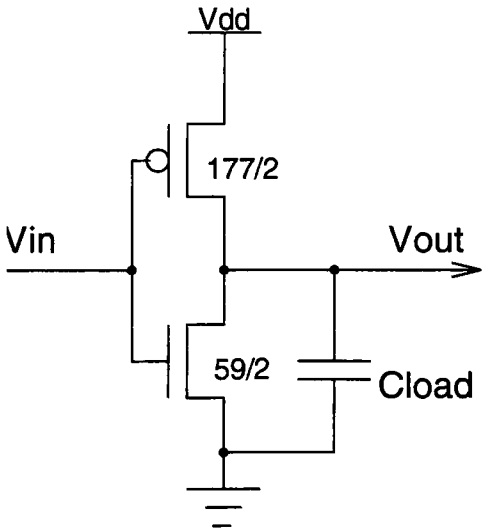


Figure 7.5: Nearly optimal evolved inverter. A stray pull down transistor is used to maintain the correct frequency response. The sizing of the transistors can still be optimized.



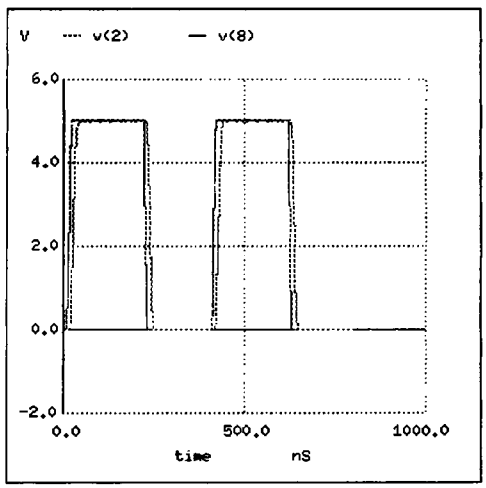


Figure 7.6: The optimal inverter. Very little improvement is possible after this.

The computed fitness is: 21.9169

Finally, in figure 7.6, the optimal inverter is found. Notice that the rise and fall times of the output match the ideal nearly perfectly. If the transistors were made any more powerful the time-domain score could be improved, but, that would increase the rise and fall times. Changing the size and fall times would disrupt the Fourier analysis score, resulting in a lower net score. The equivalent sizes for each transistor is shown. The PMOS transistor is significantly larger than the NMOS transistor. The ratio here is congruent with the target manufacturing process. The run is automatically terminated when the score hits a threshold level. For all the testing done in this section, the threshold level was set at 20.

Statistical data is recorded every time one of the deme completes a step. The best individual from each population is kept by the master process. From this collection of individuals, the best, average, and worst score is calculated. Figure 7.7 is a plot of this data from a typical run. This run was performed on a virtual machine of size 8, and a global population of 3,200 individuals. Each deme had a population size of 400.

7.1.1 Effectiveness of Parallelization

In addition to the generational data, system performance was continuously monitored and recorded. Once every five seconds the following data is recored: the total number of circuit evaluations, overall average number of evaluations per second, and the average over the last five seconds. This data can be used to determine the effectiveness of the parallelization. The inverter discussed above was evolved many times on a parallel machine of varying sizes.

Figure 7.8 shows a plot of the number of circuit evaluations (simulations) required to find an inverter which has a score under 20.0 at different levels of parallelization. The size of the global population was kept constant while the number of demes was varied. This plot shows that the GA was not significantly hampered by increasing the machine size, indicating that there was sufficient communication between the deme populations.

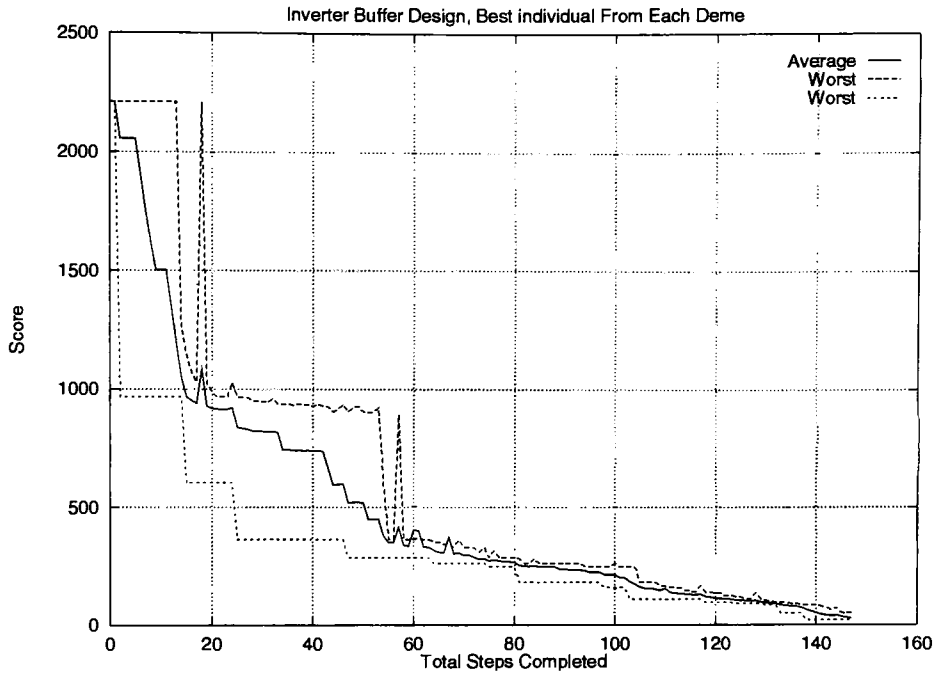


Figure 7.7: Generational data from inverter evolution.

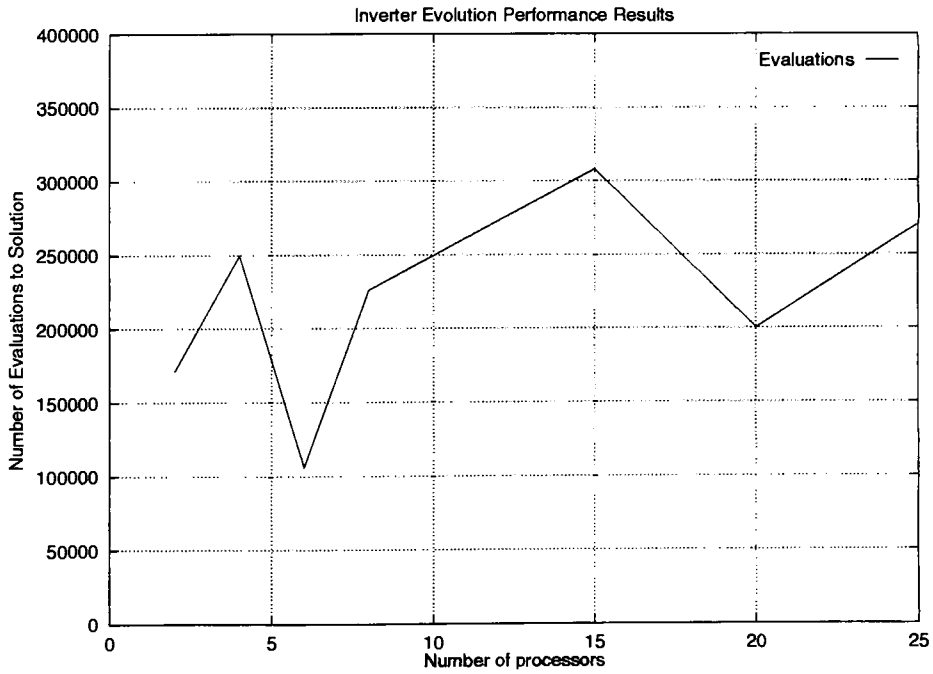


Figure 7.8: Evaluations to find an optimal inverter at different levels of parallelization

Figure 7.9 shows a plot of the number migration events which occurred during the run. There is a almost linear increase in the number of migrations with the size of the virtual machine. This is to be expected because a migration occurs after each step. With n processors, the global population is divided n ways. Figure 7.8 indicated that the number of circuit evaluations is approximately constant with n. The number of evaluations in a step is then approximately equal to the total number of evaluations divided by the number of processors. The total number of evaluations is E, the number of evaluations in a step is $e = E/N$, the number of steps is N. The number of steps can be estimated with these relationships:

$$e = \frac{E}{N}$$

$$Dk = e$$

$$D = \frac{p}{n}$$

$$\frac{E}{N} = \frac{p}{n}k$$

$$N = \frac{En}{kp}$$

where k is the number of generations in a step, p is the global population size, and D is the deme population size. For the experiments presented here, the value of k is 5 and p is 3,200. To increase communication, k can be decreased.

Figure 7.10 graphs the time it took to find the solution at different levels of parallelism. Because the total number of evaluations did not significantly increase with the number of processors, a nearly linear speedup can be expected. The communication overhead is not significant in these runs. Variations in speedup are due to the processors being loaded with other jobs. More tests are required at higher levels of parallelism in order to determine the point at which communication overhead starts to interfere with the processing time.

The average number of evaluations per second is graphed in figure 7.11. This clearly shows that the machine is not being limited by communication resources. Each node is

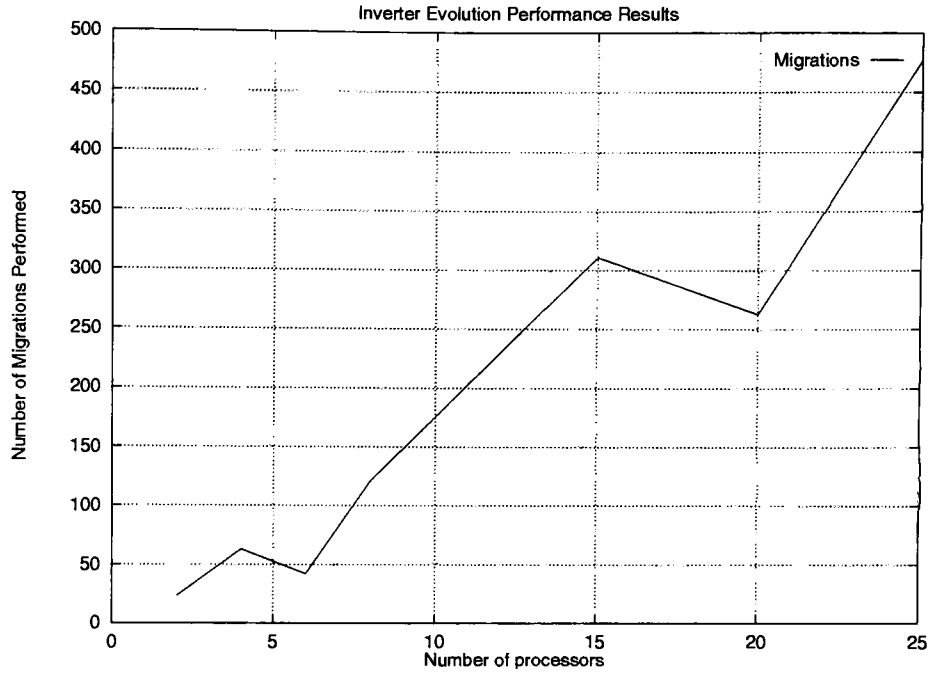


Figure 7.9: Number of migrations at different levels of parallelization

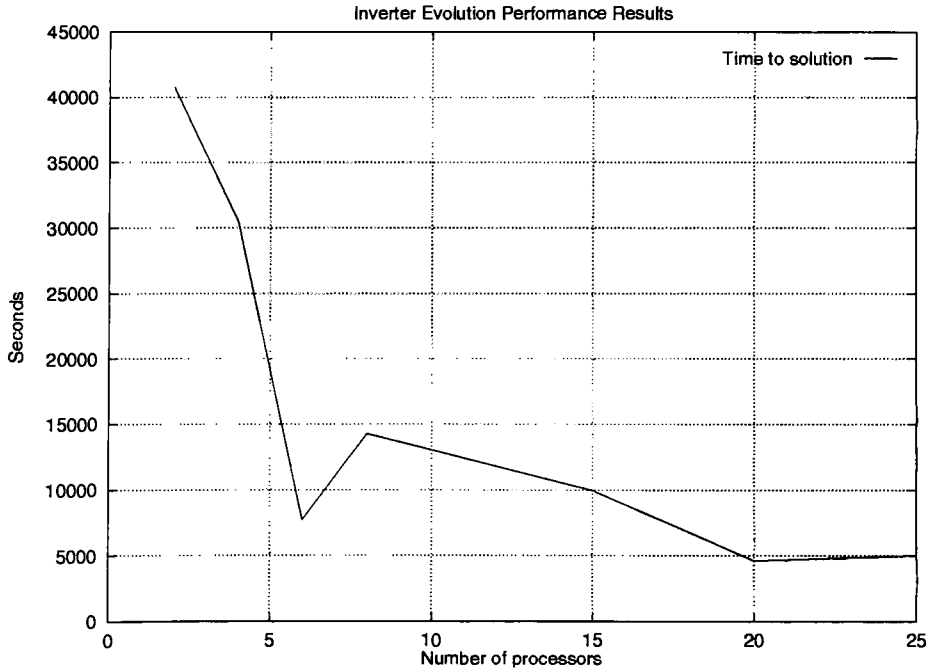


Figure 7.10: Time to solution at different levels of parallelism

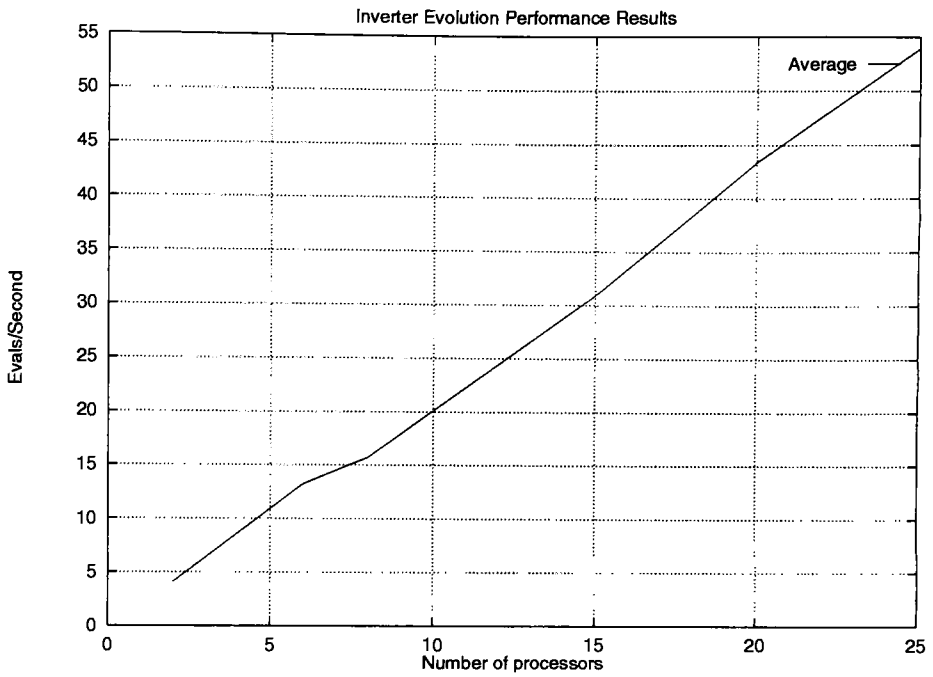


Figure 7.11: Evaluations per second at different levels of parallelism

continuously running at maximum speed.

7.2 Optimizing Transistors in an Inverter-Buffer Chain

As discussed in section 6.2.1, a series of inverters is often used in place of a single inverter for maximum speed. Each inverter in the chain is made larger and more powerful than the previous. The step size between stages is largely determined by the technology used to implement the transistors.

Figure 7.12 shows an inverter buffer chain test circuit. There are three stages in this chain, the first of which is being driven by an inverter. The inverter establishes the input conditions to match what would be found in an actual circuit where a weak signal needs to drive a heavy load. For example, there may be a situation in which a simple logic gate has to drive a high fan-out clock line that runs all across the chip. Another example is when the logic gate drives the output pins of a chip.

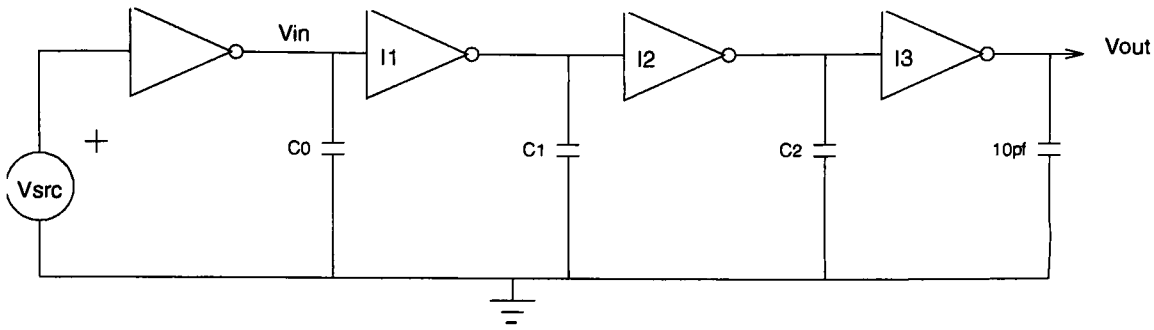


Figure 7.12: Inverter buffer chain test circuit.

Here is the seed circuit which was given to the GA for optimization, note that the output stage is doubled up for higher current drive:

```
Buffer Chain Optimization
.options acct abstol=10n  vntol=10n
.tran 1ns 600ns

M1 5 6 2 5 CMOSP
M2 0 6 2 0 CMOSN
M3 5 6 2 5 CMOSP
M4 0 6 2 0 CMOSN
M5 5 4 6 5 CMOSP
M6 0 4 6 0 CMOSN
M7 5 1 4 5 CMOSP
M8 0 1 4 0 CMOSN

*Start Footer circuit
m11 5 3 1 5 CMOSP l=1u w=8u
m12 0 3 1 0 CMOSN l=1u w=4u
r100 5 2 5Meg
r101 0 2 5Meg
r102 0 1 5Meg
r103 5 1 5Meg
r105 6 5 5Meg
r106 6 0 5Meg
r107 4 5 5Meg
r108 4 0 5Meg
r101 0 2 20K
cc 2 0 10pf
vin 3 0 dc 0 pulse(5 0 10ns 5ns 5ns 150ns 300ns)
v2 8 0 dc 0 pulse(5 0 10ns 5ns 5ns 150ns 300ns)
vccp 5 0 dc +5
.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=1
+ VTO=0.6474 DELTA=1.6230E+00 LD=5.8150E-09 KP=8.0236E-05
+ UO=729.6 THETA=1.2540E-01 RSH=9.0910E-02 GAMMA=0.5999
+ NSUB=1.3110E+16 NFS=6.5000E+11 VMAX=2.1000E+05 ETA=9.9760E-02
+ KAPPA=1.5680E-01 CGDO=9.5924E-12 CGSO=9.5924E-12
+ CGBO=2.9552E-10 CJ=2.84E-04 MJ=0.517 CJSW=1.97E-10
+ MJSW=0.100 PB=0.99
```

```
.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=-1
+ VTO=-0.8483 DELTA=1.8430E+00 LD=1.0280E-09 KP=1.9212E-05
+ UO=174.7 THETA=7.7780E-02 RSH=1.0500E-01 GAMMA=0.3635
+ NSUB=4.8130E+15 NFS=6.5000E+11 VMAX=4.0030E+05 ETA=1.3040E-01
+ KAPPA=9.9940E+00 CGDO=1.6958E-12 CGSO=1.6958E-12
+ CGBO=3.1527E-10 CJ=3.02E-04 MJ=0.497 CJSW=2.59E-10
+ MJSW=0.100 PB=0.99

.print tran v(2) v(8)
.five 6.666Meg v(2) v(8)
.eval  tran v(2) v(8)
```

The GA was able to successfully optimize the circuit for maximum performance. Here is an example of one of the best solutions:

```
M0 5 6 2 5 CMOSP l=0.8u w=19u
M1 0 6 2 0 CMOSN l=1u w=17.6u
M2 5 6 2 5 CMOSP l=0.8u w=20u
M3 0 6 2 0 CMOSN l=0.8u w=19u
M4 5 4 6 5 CMOSP l=0.8u w=6u
M5 0 4 6 0 CMOSN l=1u w=3.8u
M6 5 1 4 5 CMOSP l=1.4u w=10.4u
M7 0 1 4 0 CMOSN l=2u w=4.6u
```

Note that the technology was changed to a sub-micron manufacturing process. The maximum permitted transistor width was set to 20, and the minimum was set to 0.8. Table 7.1 shows the calculated transistor width/length ratios. The restrictions placed on the maximum transistor width made for some interesting optimizations. The first stage over-compensates for the PMOS side of the inverter. The rise time after the first stage is faster than the fall time. The second stage uses a more normal ratio of approximately 2 for the p/n ratio. This translates the faster rise time from the first stage into a fast fall time for the second stage. Now, at the output stage, the p/n ratio is almost 1. This is because of the restriction of the maximum width of 20. Both the PMOS and the NMOS transistors are constructed of the maximum possible size. The faster fall time from the second stage output translates into a faster rise time at the output stage. This equalized

the fall/rise time despite the equal transistor sizes. This is a very good optimization of the sort a conventional designer probably would not have considered.

Table 7.1: Buffer chain optimization transistor ratios

	p	n	p/n
stage 1	7.4	2.3	3.2
stage 2	7.5	3.8	1.97
stage 3	48.75	41.35	1.17

7.3 CMOS Operational Amplifier Design

An existing CMOS op-amp was used as a seed circuit for the GA:

```
CMOS Operational Amplifier Optimization
* .options acct abstol=10n  vntol=10n
* .tran 1ns 600ns
.options  tnom=27 newton
.tran 1e-07 0.00005 0

M1 13 11 11 5 CMOSP
M2 14 13 13 5 CMOSP
M3 5 14 14 5 CMOSP
M4 5 15 16 5 CMOSP
M5 5 15 15 5 CMOSP
M6 5 16 17 5 CMOSP
M7 2 11 10 10 CMOSN
M8 5 17 2 10 CMOSN
M9 15 1 18 10 CMOSN
M10 16 12 18 10 CMOSN
M11 18 11 10 10 CMOSN
M12 17 11 10 10 CMOSN
M13 11 11 10 10 CMOSN
Vp 10 0 -5
VCC 5 0 DC 5
v1 3 0 DC 0 SIN 0 0.05 100000 0 0
v2 1 0 DC 0
v3 8 0 DC 0 SIN 0 4 100000 0 0
R1 2 0  10K
R2 3 12  100
R3 12 2  85K

.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=1
+ VTO=0.6474 DELTA=1.6230E+00 LD=5.8150E-09 KP=8.0236E-05
+ UO=729.6 THETA=1.2540E-01 RSH=9.0910E-02 GAMMA=0.5999
+ NSUB=1.3110E+16 NFS=6.5000E+11 VMAX=2.1000E+05 ETA=9.9760E-02
```

```
+ KAPPA=1.5680E-01 CGDO=9.5924E-12 CGSO=9.5924E-12
+ CGBO=2.9552E-10 CJ=2.84E-04 MJ=0.517 CJSW=1.97E-10
+ MJSW=0.100 PB=0.99

.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=3.1400E-08 XJ=0.200000U TPG=-1
+ VTO=-0.8483 DELTA=1.8430E+00 LD=1.0280E-09 KP=1.9212E-05
+ UO=174.7 THETA=7.7780E-02 RSH=1.0500E-01 GAMMA=0.3635
+ NSUB=4.8130E+15 NFS=6.5000E+11 VMAX=4.0030E+05 ETA=1.3040E-01
+ KAPPA=9.9940E+00 CGDO=1.6958E-12 CGSO=1.6958E-12
+ CGBO=3.1527E-10 CJ=3.02E-04 MJ=0.497 CJSW=2.59E-10
+ MJSW=0.100 PB=0.99

.print tran v(2) v(8)
.five 100k v(2) v(8)
.eval  tran v(2) v(8)
.end
```

The test circuit consists of applying a 100 KHz, 5 millivolt signal at $V1$ and grounding the other input. The output requirements specify a 10K resistor as a load, and a gain of 800. A feedback resistor is included to stabilize the circuit. A complete optimization of this circuit was not possible due to some liquid technical problems with the hardware. A flood in some of the computer labs disabled a major portion of the nodes in the parallel machine. Initial testing revealed that this is a tough nut to crack. On a 15 processor system, after an hour, the op-amp yielded a gain of only about 10.

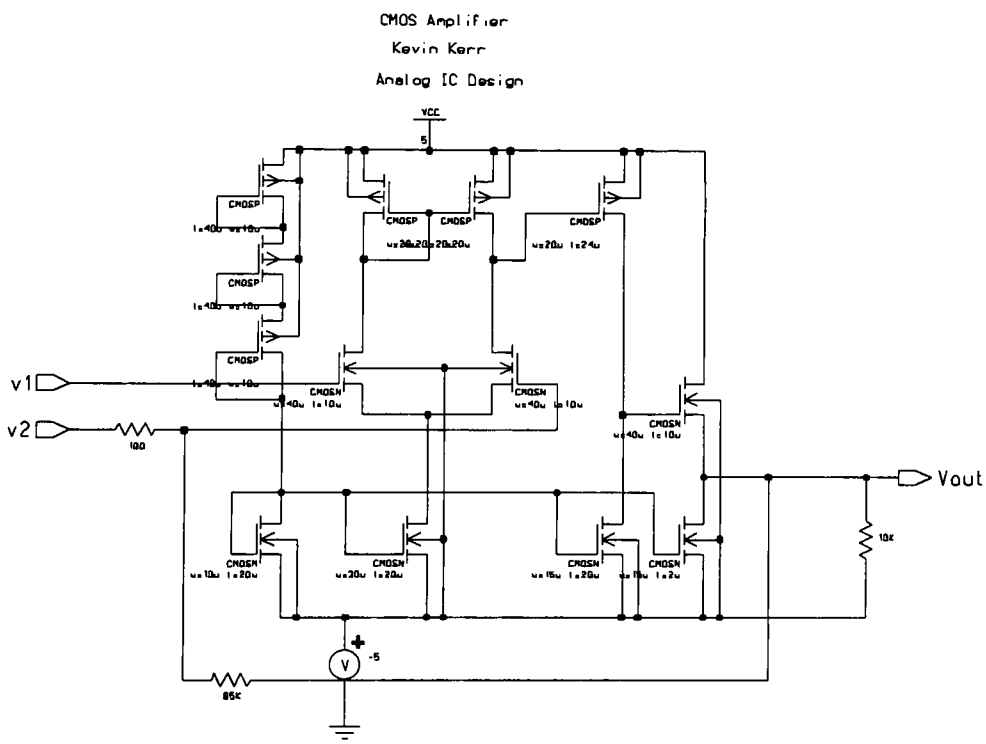


Figure 7.13: An existing operational amplifier design.

Chapter 8

Conclusion

8.1 Summary

The primary goals in this thesis were to create a system in which a genetic algorithm can be employed to design and optimize electronic circuits. In order to make this task feasible, a parallel architecture was required. There are three primary aspects to this research: a genetic algorithm to evolve circuits, an efficient parallel implementation, and a method of testing the circuits as they evolve.

The genome which the GA manipulates is modeled after a SPICE net list. This makes conversion between phenotype and genotype easy (net list to individual in population conversion). It also is also allows for any arbitrary circuit configuration, and does not put a limit to the possible connectivity of the circuit. A hierarchical genome structure is used to maximize flexibility. Because of the software interface defined by the genome, a complex sub-circuit can be included in this hierarchy transparently. As a result of this structure, just about any SPICE net list could potentially be used as a starting seed for the GA. The GA could then optimize the circuit according to the user requirements.

A parallel implementation of the GA is required. This is because circuit simulation is time consuming, and the GA needs to test large numbers of circuits. A survey of the common parallel GA architectures resulted in selecting a coarse-grained parallel GA. Initial testing with this architecture showed that it works well for a low level of parallelism. Converting

the architecture into an asynchronous implementation solved most of the problems at higher levels of parallelism, and greatly increased efficiency.

Modifications were made to a SPICE simulator in order test the circuits being evolved. Initially, a time domain analysis was used to compare the actual circuit output with the ideal output. This works well in may cases, but a better solution was found. A frequency domain analysis used in conjunction with the time domain analysis allows a wider range of circuits to be successfully evolved.

Testing resulted in the successful evolution of an optimized CMOS inverter from scratch. Fixed topology optimization was performed on a buffer chain circuit and a CMOS amplifier.

8.2 Potential Applications

The computational requirements, even for simple circuits, may rule out this approach for many applications. However, there are some specialized areas which may be suitable. One of the most interesting applications which could come out of this work is in searching for worst case scenarios. The manufacturing process for circuit components is not exact. This means that one can not precisely predict how every circuit manufactured will perform under all operating conditions. The software developed for this thesis could be utilized to search for the worst case circuit performance by encoding all of the process and environmental variables into the genome. The GA could then be configured to search for the *worst* possible performing circuit, given the manufacturing tolerances and the range of environmental conditions. Currently this type of search is done with a monte-carlo analysis which is equivalent to a random search.

8.3 Lessons Learned

8.3.1 Genome Representation

The Initially proposed format for the genome was a binary string with bit fields representing node numbers and circuit element attributes. Initial testing showed that this worked, but not well. The format would allow too many invalid circuit elements. It was difficult to specify a valid set of nodes which the GA should explore, and also specify a valid range of attributes, which may vary depending on the type of circuit element. The development of the object oriented genome, which uses allele sets, greatly improved both the flexibility and performance of the GA.

After working with the GA for a while, it seems that a more structured approach to building circuits may be advantageous. The current genome representation may have too much epistasis. This prevents good building blocks from combining in useful ways. The approach used by Koza in his circuit synthesis work helps to overcome this to a certain extent. It would not be very difficult to convert the list form currently used in the GA into a tree form. At this point it would almost be a genetic program. It may be an interesting experiment to try.

8.3.2 Asynchronous Parallelism

As clearly demonstrated in section 4.2.5, an asynchronous parallel GA is necessary for this application. The fact that other people may be running processes on some of the nodes, and that some of the nodes may have different processors, ensures that it is a waste of time to try and synchronize after each epoch (after all deme complete their step). The asynchronous approach used here did a good job of solving the problem. A different approach, which was not tried, is to perform a wall-clock synchronization. This probably would have worked, but network traffic could cause delays if everyone is trying to migrate at the same time.

8.3.3 Circuit Evaluation

The proposed circuit evaluation technique was adequate for most circuits, however, the addition of a frequency-domain analysis was beneficial. This was especially true for the free topology inverter problem. The frequency-domain analysis would immediately pick out individuals who had any kind of sharp effects on the output node voltage. This helped speed the search for the correct topology and then optimize for transistor sizing.

8.4 Concluding Remarks

This thesis represents the successful integration of a number of the important areas of computer engineering: circuit/VLSI design, parallel computing, and computer science. This is what makes things interesting. The challenge was to get it to work at all. The hope was that it would come up with something unusual. If fact, the results from the inverter buffer chain optimization were surprising. It was an unexpected, unique, and effective solution. There are many more complex, and difficult to analyze, circuits which could be considerably improved with this type of optimization.

There is a number of paths which can be taken to continue this research. The performance of the genetic algorithm may be greatly improved with the utilization of an adaptive algorithm to compensate for the different phases of evolution. Better results may come from a better genome representation. The software architecture will easily support a hybrid parallel genetic algorithm. This could vastly increase the maximum number of nodes in the parallel system. An application to dynamically change the circuit requirements on the fly could make the software more useful.

Bibliography

[AH95] Bashir Al-Hashimi. *The Art of Simulation Using PSpice: Analog and Digital.* CRC-Press, 1995.

[BC89] Mark F. Bramlette and Rod Cusic. A comparative evaluation of search methods applied to parametric design of aircraft. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 213–218. Morgan Kaufmann Publishers, 1989.

[CHMR87] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated equilibria: A parallel genetic algorithm. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148–154. Lawrence Erlbaum Associates, 1987.

[CMR91] J. P. Cohoon, W. N. Martin, and D. Richards. A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 244–248. Morgan Kaufmann Publishers, 1991.

[CP97] Erick Cantú-Paz. A survey of parallel genetic algorithms. IlliGAL Technical Report 97003, University of Illinois at Urbana-Champaign, May 1997. http://gal4.ge.uiuc.edu/illigal.home.html.

[CPG96] Erick Cantú-Paz and David E. Goldberg. Predicting speedups of ideal bounding cases of parallel genetic algorithms. IlliGAL Technical Report 96008, University of Illinois at Urbana-Champaign, December 1996. http://gal4.ge.uiuc.edu/illigal.home.html.

[Enc] Encore. Encore. Excelent refrence on the web: ftp://ftp.cs.wayne.edu/pub/EC /Welcome.html.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[GM93] Paul R. Gray and Robert G. Meyer. *Analysis and Design of Analog Integrated Circuits (3rd ed.).* Wiley, 1993.

[Gol89a] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[Gol89b] David E. Goldberg. Sizing populations for serial and parallel genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann Publishers, 1989.

[GR94] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, Jan 1994.

[HK94] Anthony M. Hill and Sung-Mo Kang. Genetic algorithm based design optimization of CMOS VLSI circuits. In Yuval Davidor, Hans-Paul Schwefeld, and Reinhard Männer, editors, *Parallel Problem Solving from Nature - PPSN III*, pages 546–555. Springer-Verlag, 1994.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial System: an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press, 1975.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[JQN+] B. Johnson, T. Quarles, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli. Spice3 version 3e user's manual. Available from http://www-cad.eecs.berkeley.edu/Software/software.html.

[KBA+97] John R. Koza, Forrest H Bennett, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1, July 1997.

[Kie94] Ron Kielkowski. *Inside SPICE: Overcoming the Obstacles of Circuit Simulation.* McGraw-Hill, 1994.

[Koz94] John R. Koza. An introduction to genetic programming. In Kenneth E. Kinnear Jr., editor, *Advances in Genetic Programming.* MIT Press, 1994.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[PL89] Chrisila B. Petty and Michael R. Leuze. A theoretical investigation of a parallel genetic algorithm. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 398–405. Morgan Kaufmann Publishers, 1989.

[PLG87] Chrisila B. Petty, Michael R. Leuze, and John J. Grefenstette. A parallel genetic algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and their*

Applications: Proceedings of the Second International Conference on Genetic Algorithms, pages 155–161. Lawrence Erlbaum Associates, 1987.

[PVM] PVM. Available from the internet at http://www.epm.ornl.gov/pvm/.

[SCC96] M. S. Selig and V. L. Coverstone-Carroll. Application of a genetic algorithm to wind turbine design. *Journal of Energy Resources Technology*, 118, March 1996.

[SV96] Volker Schnecke and Oliver Vornberger. An adaptive parallel genetic algorithm for VLSI-layout optimization. In *International Conference on Evolutionary Computation - The 4th International Conference on Parallel Problem Solving from Nature*, 1996.

[Tho96] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In *Proceedings of the First International Conference on Evolvable Sytems (ICES96)*, 1996.

[Tho97] Adrian Thompson. Temperature in natural and artificial systems. In P. Husbands and I. Harvey, editors, *Proceedings of the 4th European Conference on Artifical Life (ECAL97)*, pages 388–397. MIT Press, 1997.

[Vos94] Michael D. Vose. Correspondence on the subject of convergence. GA digest, ftp://ftp.aic.nrl.navy.mil/pub/galist/digests/v8n22, 1994.

[Wal96] Matthew Wall. GAlib. Available from the internet at http://lancet.mit.edu/ga/, 1996.

[WE93] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design:A systems perspective.* Addison-Wesley, 1993.

[ZVP] R. Zebulum, M. Vellasco, and M. Pacheco. Comparison of different evolutionary methodologies applied to electronic filter design. Accepted for the IEEE International Conference on Evolutionary Computation (ICEC98), Anchorage, Alaska, May 4-6, 1998.

[ZVP96] R. Zebulum, M. Vellasco, and M. Pacheco. Evolvable systems in hardware design: Taxonomy, survey and applications. In *Proceedings of the First International Conference on Evolvable Systems (ICES96)*, 1996.