VLSI Circuit Synthesis using a Parallel Genetic Algorithm

Mike Davis, Luoping Liu, and John G. Elias

Department of Electrical Engineering
University of Delaware
Newark, DE. 19716

ABSTRACT

A parallel implementation of a genetic algorithm used to evolve simple analog VLSI circuits is described. The parallel computer system consisted of twenty distributed SPARC workstations whose computational activity is controlled by the parallel environment coordination language Linda. Work-in-progress on using the parallel GA to realize optimized circuits and to discover new types of equivalent-function circuits is presented. The use of biologically inspired development rules to limit the scope of circuits generated by recombination operators to circuits that have an increased chance of surviving is briefly discussed.

INTRODUCTION

Our research efforts are directed towards using parallel genetic algorithms (GAs) to evolve optimized versions of analog and digital circuits from their functional descriptions. In addition, we are assessing the efficacy of using evolutionary computational methods to discover new and previously unknown types of circuit implementations that have equivalent function but different, and possibly improved, architectural features compared to their engineered counterparts.

With parallel GAs, one must consider whether to use a single large population with central or coordinated access granted to each GA, or to use a number of smaller subpopulations for each GA to operate on independently. With a single population, global variables and access contention become potential bottlenecks as each GA tries to update critical information or to access individuals in the population. With multiple subpopulations, stochastic population growth, independent evolutionary trajectories, and the need to communicate and interact at various times between the gene pools become important considerations [BIAN92].

In this paper, we first describe our parallel computer system and present initial comparative results. We then present experimental results on evolving a simple analog function from its behavioral description. We conclude by briefly discussing the limitations of unconstrained recombination and mutation operators and, as a possible solution, introduce the use of biologically inspired development rules.

EXPERIMENTAL PARALLEL COMPUTER SYSTEM

In this section, we compare and evaluate three forms of a coarse grain parallel GA designed for circuit synthesis and optimization. This work focuses on three aspects of parallel GA's that are of little importance with non-parallel implementations: population split, distributed or centralized population state management, and dissemination of population state information. The parallel GAs in this study are programmed to create analog circuit implementations that match both a behavioral description (e.g., amplifier, multiplier, etc.) as well as specific physical parameters (e.g., gain, offset, power dissipation, footprint, etc.).

Linda Coordination Language

Control of our parallel GAs is effected through the Linda coordination language, which extends the syntax of conventional programming languages to support parallel operating processes [Carr91]. The most important feature of Linda is its shared-memory programming model that links a wide range of physical computing platforms into a coherent computing structure. These platforms may range from a multiprocessor having a shared-memory system already, to a network of workstations with independent physical memory and no common address space. The Linda shared-memory model compares well to other parallel environments, whether on a physical shared-memory machine or a message-passing distributed system.

The shared-memory programming model used by Linda consists of a so-called Tuple-Space (TS), which is logically a single memory but physically may be widely distributed over the processors in the system. The TS is a virtual shared-memory data storage area that processes can read and write. C-language Linda consists of four programming language extensions: out() to move data into TS; rd() to get a copy of data from TS; in() to move data out of TS; and eval() to add a new C procedure to the TS which joins the ranks of executing processes. Data are read from the TS in an associative matching scheme. Matching criteria is sent to the TS via the in() or rd() calls, and the first tuple to match is returned to the requesting process. This associative matching leads to a simple form of load balancing where processes continuously probe the TS looking for work to do. The main effect is that equivalent processes running on faster processors end up doing more work than their slower speed counter parts.

The parallel GA used in this study is mapped onto the TS using two different schemes as shown in Figure 1. The Centralized scheme, Figure 1a, consists of a single breeding population, while the Distributed scheme, Figure 1b, makes use of multiple breeding populations that may interact with each other from time to time. In both schemes, the TS contains four types of tuples: 1) members of the breeding population, $C_{i,p}$; 2) offspring of breeding pairs, $O_{j,p}$; 3) average fitness of the population, AF_p ; and 4) best individual in the population, BI_p. The subscript, p, represents a particular breeding population in TS.

The members of a breeding population, $C_{i,p}$, are tuple data elements each of which completely describes a particular circuit (e.g., transistor parameters: type, number, dimensions, placement, and connections). In each scheme, the GA operators function as reported previously [ELIA92]. Breeding individuals, $C_{i,p}$, are randomly selected two at a time from the population by a Worker process using the Linda rd() function. The selected individuals are recombined using two crossover locations and may also undergo mutation. Two offspring are produce and evaluated. The better performing offspring's fitness value is compared to the average fitness, AF_p , which is read from TS by the Worker. If the offspring's fitness is larger than the population average its circuit description, $O_{j,p}$, is written to TS using the Linda out() function. The Worker then selects two more individuals randomly and repeats the procedure. This process is executed by each Worker (sixteen in this study) in parallel. The offspring, $O_{j,p}$, stored in TS by Worker processes do not immediately become members of the breeding population. Each new offspring must wait a short while until its Master process incorporates it into the population (explained below).

Master processes create Worker processes, initialize and manage the population, and keep track of its population state variables, best-individual (BI_p) and average-fitness (AF_p). The Master process is responsible for adding newly created offspring to the breeding population. In the present study, the population size remains constant. In this case, the Master selects randomly a member of the population from a local memory that holds only the individual's fitness value whenever a Worker writes an offspring, $O_{j,p}$, to TS. The fitness of the selected individual is compared to the average fitness, AF_p , and if less than the average the individual, $C_{i,p}$, is extracted from TS using in() and replaced by the new offspring. If the selected individual's fitness is above average, then another selection is made from the local fitness memory until an individual with below average fitness is found.

The Distributed scheme, Figure 1b, divides TS into four equivalent regions, each of which has exactly the same tuple arrangements as in the Centralized scheme. The total number of individuals is the same in both schemes, but in the Distributed scheme, each subpopulation has one Master and four Worker processes. The subpopulations interact according to a programmable schedule and exchange selected members of their respective populations.

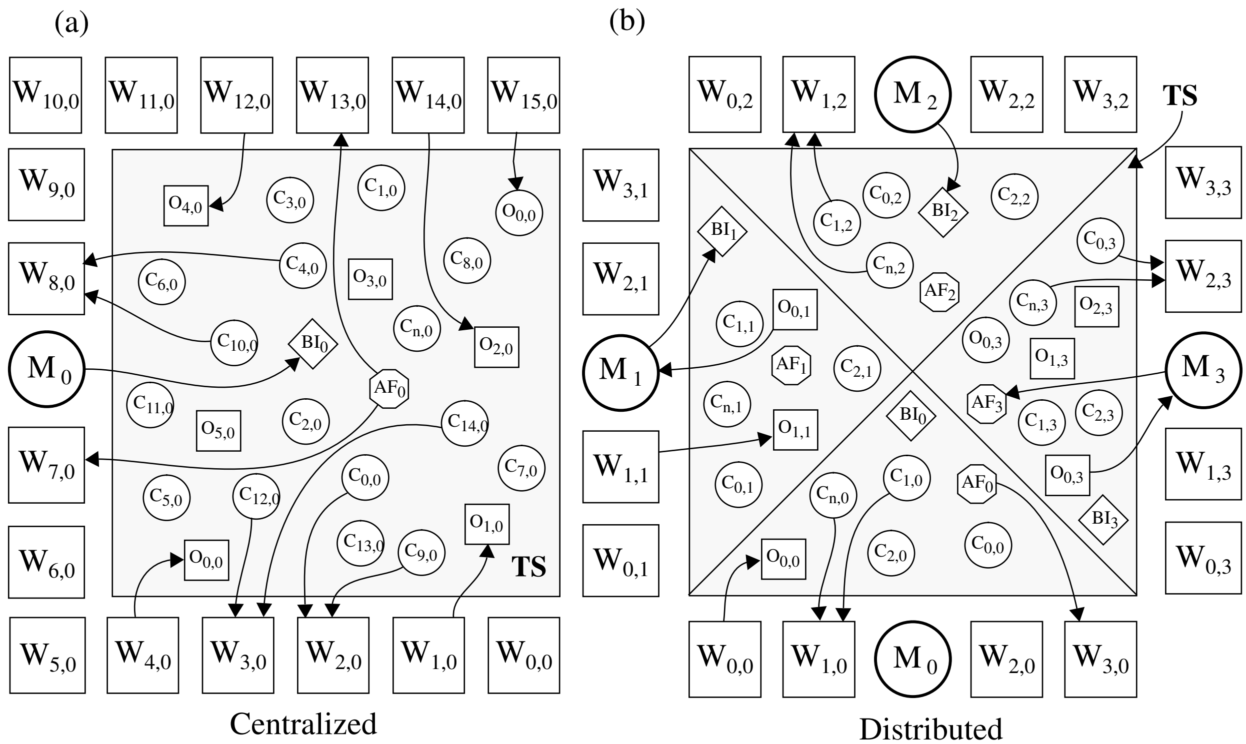

FIGURE 1. Two schemes for mapping breeding populations into tuple space (TS). Worker processes, $W_{i,p}$, select breeding pairs from population p, execute recombination and mutation operators, evaluate offspring, and place a single offspring, $O_{j,p}$ into TS if it performs better than the average individual in the population, AF_p. Master processes, M_p , compute the average fitness, AF_p, keep track of the best individual, BI_p, and are responsible for replacing below average individuals with above average offspring. In the Centralized scheme, (a), sixteen Workers access a single population. In the Distributed scheme, (b), TS is divided into four subpopulations, each operated on by one Master and its four Workers. The subpopulations interact according to a programmable schedule.

Results

For the work reported here, the computing environment consisted of a network of twenty Sun IPC workstations running a commercial Linda package. Each 25 MHz IPC had 24 MB of RAM and was connected to the other workstations through a 10 Mbps Ethernet or though a 100 Mbps FDDI network. The published performance of these machines is SPECint92 = 13.8, SPECfp92 = 11.1 or historical rating of 17.8 MIPS and 1.8 MFLOPS. Sixteen of the machines were connected to the FDDI network as well as to the Ethernet. In this study, Worker processes ran on FDDI-connected machines while Master processes ran on the remaining non FDDI-connected machines.

The GAs were set up to evolve a simple digital test circuit comprising five transistors (chromosome was 42 bits long). The evaluation of the fitness for each test circuit was programmed to take exactly one second in order to make accurate comparisons. Table 1 shows some of our results for three versions of parallel GA's compared to the non-parallel GA. The total population size in all cases was set at 1000.

Two measures of speedup with respect to the non-parallel GA are reported: 1) Speedup computed from the time taken to reach a specified fitness value; and 2) Speedup computed from the time taken to reach a specified number of generations. In both cases, speedup was calculated by dividing the time-to-reach-criterion for the non-parallel GA by the time-to-reach-criterion for the parallel GAs. The numbers reported for the parallel algorithms are averages of five separate runs. Since Linda runs on asynchronous processors, the ordering of operations taken across all machines will be different from run to run. Therefore, each run of the parallel GAs produces slightly different results.

Table 1:

Algorithm	Speedup	Speedup	Best Fitness
	to avg_fit >=29.96 speedup/max - (Minutes:Seconds)	to gen# = 40 speedup/max - (Minutes:Seconds)	(# matching bits)
Non-parallel	1 / 1 - 691:12	1 / 1 -674:20	34
Centralized	16.5/17 - 41:56	15.2/16 - 44:11	34.6
Distributed (isolated)	-	15.4/16 - 43:44	23.2
Dist w/comm masters	17.0/20 - 40:34	15.4/16 - 43:52	36

The time taken to reach the fitness criterion is dependent on the Master/Worker framework and thus Masters are included in the calculations. The Centralized algorithm showed nearly linear speedup in each of the runs with one exception which failed to reach the fitness criterion. The distributed-with-isolated-populations GA failed to reach the fitness criterion in all runs. The Distributed-with-communicating-Masters GA reached the fitness criterion in about the same time as did the Centralized GA, but its speedup was significantly less than the theoretical maximum. Hence, at least in this test, Master processes did not contribute much to completing the task. The time to reach a specific generation is mostly related to Workers processes, thus the theoretical maximum speedup was based on sixteen Workers. From Table 1, it is clear that both the Centralized and the Distributed-with-communicating-Masters schemes had about the same speedup, although not the same efficiency, and that they were able to converge to similar solutions. The Distributed-with-isolated-populations scheme was able to evaluate the same number of offspring in comparable time, but was unable to reach the same level of fitness achieved with the other schemes.

EVOLVING VLSI CIRCUITS

The complexity of analog circuits can vary from a few transistors requiring only a modest amount of computer time for evaluation to circuits comprising hundreds of transistors that puts tremendous computational demands on the evaluation phase of the GA. The number of transistors in digital circuits often out number those found in analog circuits by many orders of magnitude. While simulation of moderate-speed digital circuits is generally less demanding computationally than analog circuits, the differences become less significant as circuit operating speeds increase. In this work, we begin with very simple analog functions, and from these simple beginnings, we expect that more complex circuits can evolve.

We began by setting up the GA to evolve circuits that amplify input signals with maximum gain, minimum offset voltage, maximum voltage swing, minimum silicon footprint, and maximum common mode input voltage range for differential input signals. Here we report only on results obtained with single-ended input signals, which generally require the simplest of circuit implementations. The simplest two-transistor CMOS single-ended amplifier can be realized with four different wiring patterns, as shown in Figure 2. These are well known inverting amplifier circuits that differ in gain, voltage swing, bias voltage requirements, and silicon footprint (see Figure 3a).

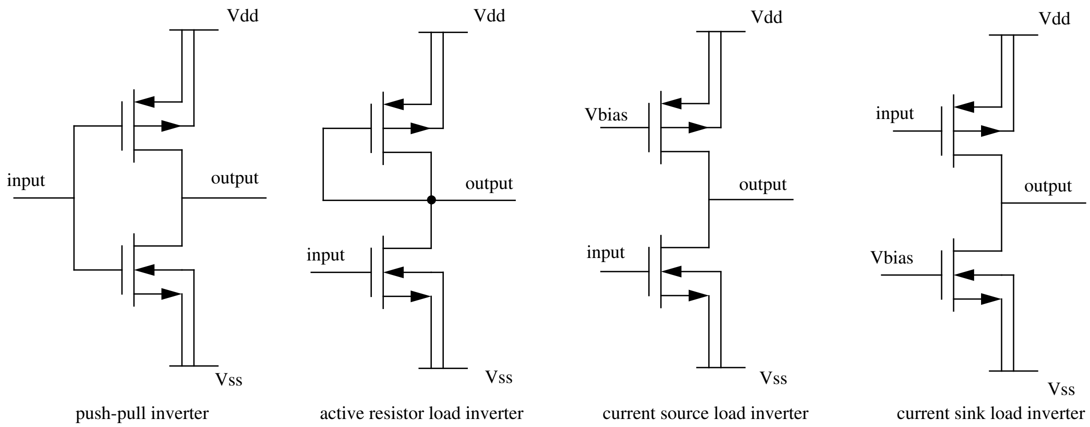

FIGURE 2. The four types of inverter amplifiers discovered by evolutionary computation. The fitness function was increased for circuits that had high gain and small silicon footprint.

The GA discovered all four types of inverter during the course of evolution. However, as shown in Figure 3b, the number of different circuit types (species) was subject to selection that favored those individuals that had the highest gain and smallest silicon footprint. Thus, as time advanced, the number of active resistor, current sink, and current source inverters diminished, eventually giving way to the push-pull inverting amplifier which became the dominant specie after generation fifteen.

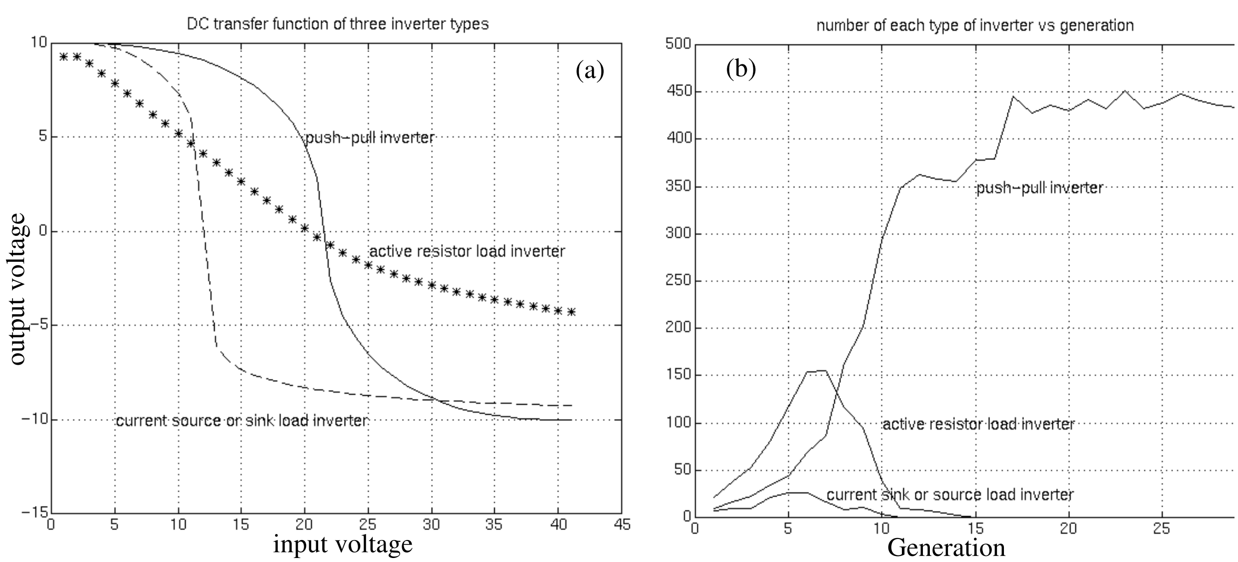

FIGURE 3. a) Input-output transfer function of the four types of inverting amplifier. b) An example of speciation. Evolution of inverter type in population as function of epoch. The active resistor, current sink, and current source inverters made up a significant part of the population through the first ten generations of evolution. After this, the dominant species became the push-pull inverter.

Development Rules

Population initialization and offspring creation present a difficulty in evolving complex circuits in which the fundamental building block is the transistor. Metal oxide field effect transistors (MOSFETs) have four terminals that connect either to the terminals of other MOSFETs or to power rails. As the number of transistors in a circuit increases the number of possible connection patterns for the circuit grows in a factorial manner. As might be expected most of these connection patterns result in non-viable circuits that can not produce offspring. This problem is most apparent if the population is initialized with random connections. In this case, all but a few individuals out of the thousands in the population are ineffective in generating viable offspring. Therefore, evolution either falters at the outset or takes an unacceptably long time to produce useful circuits. Non-viable offspring are also frequently produced from normal parents. These offspring never have a chance of joining the breeding population, but they take up precious compute resources during their evaluation.

To deal with these problems we have adopted a scheme inspired by developmental processes that occur in nature. We apply rules of development [Nort94] whenever a circuit is created during population initialization. Application of the development rules has the effect of limiting the range of possible individuals to those that have a reasonable chance of surviving and reproducing. The development rules we are evaluating are based on layout procedures that VLSI circuit designers routinely employ (e.g., inter-transistor connections depend on distance between transistors, transistor terminal connections depend on each other, transistor sizing, etc.). During recombination and mutation, offspring are checked to verify compliance with the development rules. Those not in compliance are discarded before they can be evaluated, thus saving computer resources. The application of development rules does not guarantee that a particular circuit will work or even that it will ultimately give rise to successful progeny. The rules only guarantee that the connection patterns for each individual will be sensible and not be significantly unlike those found in an engineered circuit.

CONCLUSIONS

Results from initial experiments with a parallel implementation of a GA that is used to evolve VLSI circuits have been presented. Future work will evaluate the effects on convergence and speedup of varying population size, frequency of subpopulation interaction, Master/Worker task partitioning, and the number of Worker processes. In addition, we plan to refine our development rules for creating VLSI circuits and to evaluate the effects of using these rules for producing members of the initial population.

REFERENCES

[BIAN92] Bianchini, R., C. Brown. Parallel Genetic Algorithms on Distributed-Memory Architectures. Technical Report 436, Computer Science Department. University of Rochester, Rochester NY, August 1992.

[CARR91] Carriero, N., and D. Gelernter. How to Write Parallel Programs: A First Course. Massachusetts: MIT Press, 1991.

[ELIA92] J. G. Elias, “Genetic generation of connection patterns for a dynamic artificial neural network,” in *COGANN-92*, *Combinations of Genetic Algorithms and Neural Networks*, eds. L. D. Whitley and J. D. Schaffer, IEEE Computer Society Press, Los Alamitos, CA, pp 38-54, 1992

[NORT94] David P. M. Northmore and John G. Elias, “Evolving synaptic connections for a silicon neuromorph,” submitted to EC-94