An Agent Inspired Reconfigurable Computing Implementation of a Genetic Algorithm

John M. Weir and B. Earl Wells

March 14, 2003

Abstract

Many software systems have been successfully implemented using an agent paradigm which employs a number of independent entities that communicate with one another to achieve a common goal. The distributed nature of such a paradigm makes it an excellent candidate for use in high speed reconfigurable computing hardware environments such as those present in modern FPGA's. In this paper, a distributed genetic algorithm that can be applied to the agent based reconfigurable hardware model is introduced. The effectiveness of this new algorithm is evaluated by comparing the quality of the solutions found by the new algorithm with those found by traditional genetic algorithms. The performance of a reconfigurable hardware implementation of the new algorithm on an FPGA is compared to traditional single processor implementations.

1 Introduction

Since its introduction in the mid 1980s the Multiagent paradigm for constructing software systems has grown dramatically. Multiagent systems emerged from the field of Distributed Artificial Intelligence (DAI) and have become prevalent in DAI. Multiagent systems are comprised of a number of relatively simple computational entities known as agents that are capable of communicating with one another. Multiagent systems are inherently distributed and have no centralized control. Although each agent's computation is relatively simple, fairly complex behavior can emerge from when multiple agents interact with one another.

Although many competing definitions of what constitutes an agent exist there is generally good agreement that agents should operate in a system with other agents and that the agents should display the characteristics of autonomy and sociability [1]. Agents are autonomous in that they operate independently and asynchronously. Each agent makes its own decisions on actions to take and does not operate either as a master or a slave. Agents are social in that they communicate with other agents in the system to exchange data. Multiagent systems are comprised of potentially many independent, autonomous, relatively simple agents that communicate with one another and cooperate and/or compe to achieve a common goal.

2 Reconfigurable Computing Implementation of Agents

Recently, reconfigurable computing implementations of multiagent systems have been proposed [2, 3]. These reconfigurable computing agent implementations have the potential of providing systems with higher performance than multiple agent systems implemented using an instruction set processor. Agents display several characteristics that make them a good candidate for reconfigurable computing implementations.

First, each agent is usually relatively simple. This suggests that several agents could be implemented on a single FPGA. Furthermore, since it is frequently the case that all of the agents are identical to one another, the regular structure of the multiagent system also facilitates its efficient implementation on an FPGA or other reconfigurable device.

Also, multiagent systems are inherently distributed and have no centralized control and the agents communicate with each other asynchronously. This structure allows the agents to be placed on the FPGA in a manner that minimizes timing concerns. Furthermore, the distributed architecture and asynchronous communication scheme map well to a multiple FPGA device implementation if the total number of agents desired exceeds the capacity of a single device.

2.1 An Agent Inspired Genetic Algorithm

Genetic algorithms map well to a multiple agent system where each agent performs the role of a single individual in the population. In this model, the computation required of each agent is relatively simple. To implement a genetic algorithm, offspring must be formed using crossover and mutation operations and the fitness of the offspring must be computed. In a multiagent paradigm it is desirable to perform these computations locally at each agent to avoid centralized control and its associated bottlenecks. However, a standard genetic algorithm such as that presented in [4] has features that are not amenable to implementation as a multiple agent system.

Hence, the standard genetic algorithm was modified to produce an algorithm in which all decisions are made locally within the agent that implements each individual in the population. This modified genetic algorithm is called Genetic Algorithm for Reconfigurable Computing (GARC). Furthermore, the decisions for mating and population replacement require only the fitness value of the individual's mate to be received from another agent. Thus, GARC executes independently in each agent with limited information from other agents in the system. GARC utilizes a number of agents to represent each individual in the population and one additional agent called the Global Fitness Registry agent that saves the chromosome and fitness of the best solution found to date. The GARC is summarized below from the standpoint of a single agent in the system.

```
While (not Done)
   Send chromosome and fitness
   If (one or more chromosomes are received)
      Form child using crossover
   else
      Form child using mutation
   If (Child's Fitness is better than Parent's fitness)
      Replace chromosome and fitness with Child's
      Register chromosome and fitness
end while
```

In the GARC algorithm a single child will be formed by each agent for each generation. Although variations of GARC were evaluated in which the parents mated at each agent could be any two individuals in the population, the performance for the evaluated fitness functions was comparable if each agent also served as a parent with the mating chromosome coming from another individual in the population. Since the latter approach requires less communication and is simpler to implement in reconfigurable computing, the algorithm described here uses that approach. Each individual will randomly choose one or more other individuals to solicit for mating each generation. The mating solicitation occurs by sending the soliciting agent's chromosome and fitness to one or more other agents in the system. A system parameter controls how many other agents each agent will solicit for mating each generation.

For example, agent 0 may choose to solicit agent 1 for mating. Agent 0 will send its chromosome and fitness value to agent 1. If agent 0 has the best fitness value of all of the agents who solicited agent 1 for that generation then agent 1 will choose to mate with agent 0 by using the chromosome sent by agent 0 for crossover with its own chromosome to produce a single offspring. The offspring produced may also be mutated. If agent 1 is solicited by an agent with a better fitness value than agent 0 then agent 1 will not use the chromosome sent by agent 0 but will use the other agents chromosome instead. If an agent receives no chromosomes from other agents in a generation then an offspring will be produced by mutating the agent's chromosome.

Thus, in GARC, each generation of n individuals will be formed by 2n parents. Every individual in the population will be guaranteed to serve as a parent at least once every generation (each agent always uses its own chromosome for crossover or mutation). However, individuals with better fitness values will be more likely to be chosen as parents multiple times. In fact, the best individual will be guaranteed to be a parent one more time than the system parameter defining the number of mating solicitations per generation per individual (once by the individual itself and once by each of the individuals it solicited). Thus, the mating selection is biased towards individuals with better fitness values and the system parameter increases the selection pressure as it is increased. This allows GARC to implement a biased mating selection policy while each individual has limited knowledge of other individuals fitness values and with no centralized computation so each agent can make decisions locally with only the data communicated to it by other agents. This behavior is adaptable to the agent paradigm of autonomous operation with incomplete knowledge of the system.

Also, GARC implements a biased replacement policy using only local information which is a variation of binary tournament selection [13]. In binary tournament selection two individuals are selected at random from the population and the individual with the better fitness is retained for the next generation. In the version of binary tournament selection used in GARC the selection is not random and each child is entered into a tournament competing with one of its parents. Variations of GARC in which the tournament would not be between parent and child produced similar results, so the current implementation of GARC implements a binary tournament between child and parent which requires less communication and is more efficient to implement in hardware. Each agent either replaces its own chromosome with the chromosome of the offspring it formed or retains the previous generations chromosome. Again, each agent is able to act autonomously with incomplete system knowledge. Hence, the GARC algorithm conforms to the agent paradigm of autonomous entities communicating with one another and making independent decisions with incomplete knowledge of the system.

3 GARC Performance on a Conventional Computer

GARC was first implemented on a conventional computer system in the C programming language to test its performance using several fitness functions and evaluate several variations of the algorithm. The final implementation of the GARC was chosen to offer reasonable performance compared to traditional genetic algorithms while allowing for fast and efficient implementation in hardware. Although there are many variations of genetic algorithms which may offer better performance, two were chosen for comparison to GARC. The first was the simple genetic algorithm outlined in [4] and the second was a similar algorithm that implemented a biased replacement strategy.

One fitness function used to evaluate GARC was the well known ONEMIN fitness function. This fitness function is simply the number of ones in the chromosome. The chromosome used was 32 bits long. The genetic algorithms attempted to minimize the fitness function. This fitness function has no local minimums and rewards algorithms that converge quickly. Another fitness function used was the well known traveling salesman problem (TSP). The problem instance used was from [5] and was the 29 city problem known as bays29.

A number of test runs of the GARC and each of the two more traditional genetic algorithms were performed with the number of generations capped at 5000 and the population size was set to 64 and the results were averaged. A completely random search of the state space for each of the fitness functions was also executed and its performance was compared to the performance of the genetic algorithms. The number of generations required to find the optimal solution (for the ONEMIN fitness function) or the best solution found (for the TSP) were recorded and compared to gauge the performance of the algorithms.

Algorithm	Generations	Optimal
Random	5000	0
Unbiased Replacement	697	100
Biased Replacement	30	100
GARC	48	100

Table 1: ONEMIN Performance Comparison

Algorithm	Fitness	Ratio to Optimal
Random	3891	1.74
Unbiased Replacement	3395	1.52
Biased Replacement	2665	1.19
GARC	2987	1.34

Table 2: TSP Performance Comparison

3.1 GARC Performance for ONEMIN Fitness Function

The ONEMIN fitness function was used with the GARC, a random search, and the two variations of the traditional genetic algorithm (unbiased and biased replacement strategy) with a population size of 64 and number of generations capped at 5000. The mean number of generations each algorithm executed, and the percentage of runs in which an optimal solution was found are shown in Table 1.

The random search was unable to find the optimal solution in 5000 generations while the genetic algorithms found the optimal solution every time. The biased replacement policy version of the traditional genetic algorithm performed best with the GARC performing next best and the unbiased replacement policy GA performing considerably worse. It was expected that the biased replacement policy algorithm would perform better for this fitness function since it will converge faster and the ONEMIN fitness function has no local minimums. From these results it appears that GARC converges faster than the unbiased replacement policy algorithm but not as fast as the biased replacement policy algorithm.

3.2 GARC Performance for the Traveling Salesman Problem

The 29 city TSP was used as a fitness function with the GARC, a random search, and the traditional genetic algorithm with unbiased replacement policy with a population size of 64 and the number of generations capped at 5000. The 29 city traveling salesman problem (TSP) used as a fitness function had a much larger search space than the fitness functions used. Consequently, none of the algorithms were able to find an optimal solution to the TSP instance. Therefore, the performance of the algorithms was measured by comparing the average best fitness value found and the ratio of this value to the known optimal value. These results are shown in Table 2.

Again, all of the genetic algorithms performed significantly better than the random search algorithm and the GARC algorithm performed better than the unbiased replacement genetic algorithm but not as well as the biased replacement traditional genetic algorithm.

4 GARC Implementation in Reconfigurable Hardware

Once the implementation of the GARC algorithm on a traditional computer system had demonstrated that it was capable of producing reasonably good results when compared to conventional genetic algorithms it was implemented in reconfigurable hardware on a standard FPGA. The performance of this implementation was compared to the software implementation executing on a conventional computer system. The FPGA implementation was targeted to a Xilinx 2000e 560 FPGA using the Handel C hardware description language (HDL) [6]. The Handel C language compiler executed on a standard PC in which a PCI board with the Xilinx FPGA was installed. The GARC interfaced to a C program executing on the PC to report results after the run was complete. The fitness function implemented was the ONEMIN fitness function previously described. The number of agents implemented was varied from 3 to 17.

The first step in implementing the GARC in the FPGA was to determine how to take advantage of the parallelism afforded by the hardware. The implementation of the GARC included coarse grained, medium grained, and fine grained parallelism. The coarse grained parallelism was obtained by executing each agent as a separate hardware entity. Thus, each agent in the system could operate in parallel. The medium grained parallelism was obtained by executing the major sub-systems of each agent in parallel where practical. The fine-grained parallelism was obtained using the par construct of the Handel C language to execute individual HDL statements in parallel.

Each agent consisted of four separate sub-systems operating in parallel. These sub-systems were the Random Number Generator (RNG), the SendMate unit, the GetMate unit, and the Spawn unit. The Random Number Generator provided random numbers to each of the other sub-systems. The RNG is based on an additive congruent random number generator found in [7] and produces eight bit random numbers. It is capable of producing a new random number every five cycles.

The SendMate unit communicated to other agents over a blocking read, blocking write data channel provided in the Handel C hardware description language. The agents were completely connected with a dedicated communication channel from each agent in the system to every other agent in the system. This allowed for direct communication between any two agents with no delays caused by waiting on a communication channel. It would be anticipated that as the number of agents increased the routing resources of the FPGA would be exhausted and a non-completely connected inter-connect scheme would have to be used. However, for the implementation tested with up to 17 agents no such effect was observer.

The SendMate unit was responsible for sending the agents chromosome and fitness to other agents in the system to solicit mating. The SendMate unit sends the chromosome and fitness to an agent determined by the RNG output and is capable of sending the chromosome and fitness every two cycles. Delays are inserted to reduce the frequency that the chromosome and fitness are sent to approximately once per generation. The number of delays could be decreased to effectively increase the number of mating solicitations per generation and increase the mating selection pressure. The mating selection pressure can thus be tuned for different fitness functions. Too much mating selection pressure can lead to pre-mature convergence on a sub-optimal solution while too little can slow the rate of convergence increasing the number of generations required to find a good solution. The GetMate unit receives the chromosome and fitness values from other agents. The mating selection is performed in this unit. The GetMate unit keeps the best chromosome sent to the agent for each generation and provides this chromosome to the Spawn unit.

Agents	Gates (K)	Nanoseconds
3	271	135
5	454	144
9	825	190
17	1591	190

Table 3: GARC FPGA Performance Data

The Spawn unit performs the actual spawning of a child by crossover and/or mutation. If a chromosome has been received from another agent then the Spawn unit performs a crossover operation to form a child. The child formed is probabilistically mutated based on the current random number from the RNG. The Spawn unit also performs the selective replacement replacing the parent's chromosome if the child's fitness is better that the fitness of the parent.

The hardware implementation of the GARC was executed on the Xilinc FPGA with 3, 5, 9, and 17 agents. The equivalent number of gates utilized by the design, average clock cycles per generation, clock rate, and nanoseconds per generation are shown in Table 3

An interesting observation from this table is that the performance of the hardware implementation of the GARC was nearly constant as the number of agents and thus the population size increased by nearly a factor of six. The performance would be almost perfectly constant except for the fact that the maximum clock rate of the design decreases from 40 MHz to 30 MHz when the number of agents is increased from five to nine. This is probably due to longer routing traces being required as the FPGA

Agents	GARC SW	GARC HW	Speedup
3	2700	135	20
5	4600	144	32
9	8200	190	43
17	15100	190	79

Table 4: GARC FPGA Performance Data

becomes more completely utilized. Another interesting observation that may be made from this table is that the size of the design in total gates increases in a nearly linear fashion as the number of agents is increased. Thus, the performance remained nearly constant because the amount of hardware resources that were utilized increased in proportion to the number of agents that were employed.

In Table 4, the performance of the GARC is compared to the performance of a software implementation of the simple genetic algorithm and a software implementation of the GARC executing on an PowerPC G4 processor at 877 MHz. This table shows the execution time in nanoseconds per generation of the simple genetic algorithm, the GARC software implementation, and the GARC FPGA implementation, as well as the speedup of the FPGA implementation compared to the GARC software implementation also increases.

The hardware implementation of the GARC out performs the software implementation by more than an order of magnitude for all of the number of agents tested. Since the performance of the hardware implementation of the GARC is nearly constant as the number of agents increases and the performance of the software implementation increased nearly linearly as the number of agents increases the speedup of the hardware implementation compared to the software implementation increases as the number of agents is increased. As the number of agents in the system is increases the speedup is 20. It increases to nearly 80 as the number of agents is increased.

5 Comparison to Other Hardware Implementations of GA's

There have been several other hardware implementations of genetic algorithms reported in the literature. Although some are similar to the GARC in some ways the GARC has characteristics the are different from the other approaches. A hardware implementation of the compact genetic algorithm [8] displays a speedup of nearly three orders of magnitude compared to a software version. The compact genetic algorithm manipulates a single l-dimensional vector where l is the length of the chromosome instead of the entire population of chromosomes. This dramatically reduces the number of bits required to represent the population and allows the algorithm to be implemented very efficiently in hardware. However, the compact GA is not a full-fledged GA and has limited applicability.

Another hardware implementation of a GA that is completely distributed as is the GARC is presented in [10]. However, this architecture is implemented on an array of processors instead of on an FPGA. Also, this architecture has a fixed mating scheme and does not allow for any individual to mate with any other as a traditional GA does and as the GARC does.

A hardware implementation of the steady state genetic algorithm is presented in [9]. This implementation is a hardware pipeline implementation of the steady state GA and the execution time will increase linearly with the population size. The GARC's execution time is nearly constant as the population size increases. Another implementation of a GA on a standard FPGA using the Handel C language is presented in [11]. This implementation also utilizes a pipeline approach.

A hardware implementation of the traveling salesman problem is presented in [12]. However, this implementation executes on a custom reconfigurable computer and not a single FPGA. Also, this implementation is a pipeline implementation whose execution time increases linearly with the population size unlike the GARC.

6 Conclusions

The GARC is a modification of the standard genetic algorithm that was inspired by an agent approach to the genetic algorithm. The GARC allows the genetic algorithm to be executed by a number of independent autonomous agents communicating with one another. The GARC has no centralized control and no centralized location for all of the chromosomes and fitness values in the system. Instead, the data in the GARC is distributed throughout the agents with no single agent having complete knowledge of the system. This distributed data model and lack of centralized control make the GARC well suited to implementation in a reconfigurable computing environment.

The GARC algorithm performed comparably to traditional genetic algorithm when applied to candidate fitness functions. Also, the performance of the GARC on a standard FPGA was significantly better than the performance of the GARC implemented on a standard computer system. Depending on the population size the FPGA implementation executed between 20 and 79 times faster than a conventional software implementation.

Furthermore, the execution time of conventional implementations increases linearly with population size while the execution time of the implementation in reconfigurable hardware is nearly constant as the population size increases. Hence, the performance speedup of the FPGA implementation as compared to a traditional software implementation will increase as the population size increases.

The equivalent number of gates required for the implementation of the GARC on the FPGA increases as the population size increases in a nearly linear manner. Hence, as FPGA gate sizes increase more agents should be able to be placed on an FPGA resulting in higher performance speedups compared to a conventional software implementation. As device technology advances and more gates become available techniques such as GARC that scale well to utilize more hardware resources efficiently while maintaining near constant execution times will become more attractive.

References

[1] Weiss, Gerhard Editor, *Multiagent Systems A Modern Approach to Artificial Intelligence* The MIT Press, 1999

[2] Hamid R. Naji ,B. Earl Wells, M. Aborizka, "Hardware Agents", *Proceedings of the ISCA 11th International Conference on Intelligent Systems on Emerging Technologies(ICIS-2002)*,Boston, MA, July 2002

[3] Hamid R. Naji, John Weir, B. Earl Wells, "Applying the Multi-Agent Paradigm to Reconfigurable Hardware , A Sensor Fusion Example ", *Proceedings of the Second International Work Shop on Intelligent Systems Design and Application (ISDA2002)*, Atlanta ,GA, August 2002.

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

[5] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[6] http://www.celoxica.com/tech/handel-c/default.asp

[7] Knuth, Donald *The Art of Computer Programming*, Volume 2

[8] Aporntewan, C., and Chongstitvatana, P., "A Hardware Implementation of the Compact Genetic Algorithm" *Proceedings of the 2001 IEEE Congress on Evolutionary Computation*, May 27-30 2001

[9] Kim, J, Choi Y., Lee C., Chung D. "Implementation of a High-Performance Genetic Algorithm Processor for Hardware Optimization" *IEICE Transactions on Electronics* Vol E85-C NO. 1 January 2002

[10] Sekanina, L. and Dvorak, V., "A Totally Distributed Genetic Algorithm: From a Cellular System to the Mesh of Processors"

[11] Peter Martin A Pipelined Hardware Implementation of Genetic Programming using FPGAs and Handel-C. EuroGP2002, Kinsale, Ireland

[12] Graham, P. and Nelsom B. "A Hardware Genetic Algorithm for the Traveling Salesman Problem on SPLASH 2", *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications*, 1995

[13] Chambers, Lance Editor, *The Practical Handbook of genetic Algorithms Applications* Chapman and Hall, 2001