FPGA-based Implementation of Genetic Algorithm for the Traveling Salesman Problem and its Industrial Application[1]

Iouliia Skliarova, António B.Ferrari

Department of Electronics and Telecommunications
University of Aveiro, IEETA
3810-193 Aveiro, Portugal
iouliia@ua.pt, ferrari@ieeta.pt

Abstract. In this paper an adaptive distribution system for manufacturing applications is considered and examined. The system receives a set of various components at a source point and supplies these components to destination points. The objective is to minimize the total distance that has to be traveled. At each destination point some control algorithms have to be activated and each segment of motion between destination points has also to be controlled. The paper suggests a model for such a distribution system based on autonomous sub-algorithms that can further be linked hierarchically. The links are set up during execution time (during motion) with the aid of the results obtained from solving the respective traveling salesman problem (TSP) that gives a proper tour of minimal length. The paper proposes an FPGA-based solution, which integrates a specialized virtual controller implementing hierarchical control algorithms and a hardware realization of genetic algorithm for the TSP.

1 Introduction

Many practical applications require solving the TSP. Fig. 1 shows one of them. The objective is to distribute some components from the source S to destinations $d_1,...,d_n$. The distribution can be done with the aid of an automatically controlled car and the optimization task is to minimize the total distance that the car has to travel. All allowed routes between destinations are shown in Fig. 1 together with the corresponding distances $l_1,...,l_m$. Thus we have to solve a typical TSP. Note that the task can be more complicated, where more than just one car is used for example. Indeed, some routes and some destinations can be occupied, which requires the TSP to be solved during run-time. Let us assume that the car has an automatic control system that is responsible for providing any allowed motion between destinations and for performing the sequence of steps needed to unload the car in any destination point (see Fig. 1). Thus the distribution system considered implements a sequence of control sub-algorithms and in the general case this sequence is unknown before execution time. So we have an example of adaptive control for which the optimum

[1] This work was sponsored by the Portuguese Foundation of Science and Technology under grants No. FCT-PRAXIS XXI/BD/21353/99 and No. POSI/43140/CHS/2001

sequence of the required sub-algorithms has to be recognized and established during run-time.

From the definition we can see that the problem can be solved by applying the following steps:

1. Solving the TSP for a distributed system such as that is shown in Fig. 1.
2. Assembling the control algorithm from the independent sub-algorithms taking into account the results of point 1.
3. Synthesizing the control system from the control algorithm constructed in point 2.
4. Implementing the control system.
5. Repeating steps 1-4 during execution time if some initial conditions have been changed (for example, the selected route is blocked or the required destination point is occupied).

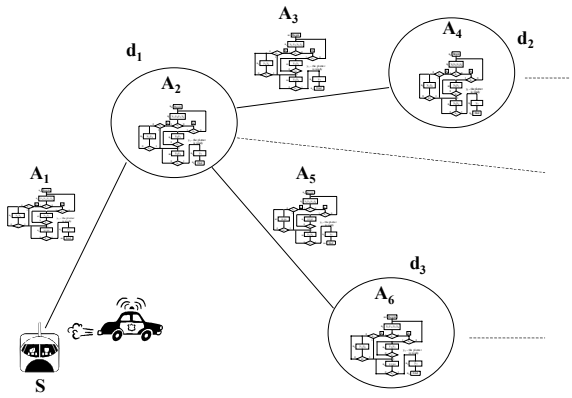

Fig. 1. Distributing system that supplies components from the source *S* to destinations $d_1,...,d_n$. The algorithms A_1, A_3, A_5 affect motions between the respective destinations and the algorithms A_2, A_4, A_6 describe unloading operations

We suggest realizing the steps considered above in an FPGA, integrating in this way the synthesis system with the implementation system. In order to solve the TSP, a genetic algorithm (GA) described in [1] has been used. Currently, only a part of the GA is implemented in FPGA. That is why we employ a reconfigurable hardware/software (RHS) model of the computations [2]. The task of point 2 can be handled with the aid of a hierarchical specification of the virtual control algorithms [3]. There are many known methods that can be used for synthesis of control systems (see point 3), for example [4]. The implementation of the control system (see point 4) was done based on the XCV812E FPGA.

The remainder of this paper is organized as follows. Section 2 provides a description of the GA employed. A hardware implementation of a part of the algorithm is considered in section 3. Section 4 gives some details about the specification of the control algorithms. The process of synthesis and implementation of virtual control circuits is presented in section 5. Section 6 contains the results of experiments. Finally, the conclusion is in section 7.

2 Genetic Algorithm for Solving the TSP

The TSP is the problem of a salesman who starts out from his hometown and wants to visit a specified set of cities, returning home at the end [5]. Each city has to be visited exactly once and, obviously, the salesman is interested in finding the shortest possible way. More formally, the problem can be presented as a complete weighted graph $G=(V, E)$, where $V=\{1, 2, ..., n\}$ is the set of vertices that correspond to cities, and E is the set of edges representing roads between the cities. Each edge $(i, j)\in E$ is assigned a weight l_{ij}, which is the distance between cities i and j. Thus the TSP problem consists in finding a shortest Hamiltonian cycle in a complete graph G. In this paper we consider the symmetric TSP, for which $l_{ij}=l_{ji}$ for every pair of cities.

The TSP is one of the best-known combinatorial optimization problems having many practical applications. Besides the distribution system described in the previous section, the problem is of great importance in such areas as X-ray crystallography [6], job scheduling [5], circuit board drilling [7], DNA mapping [8], etc. Although, the problem is quite easy to state, it is extremely difficult to solve (the TSP belongs to the class of NP-hard problems [9]). That is why many research efforts have been aimed at finding sub-optimal solutions that are often sufficient for many practical applications. Because of being NP-hard, having a large solution space and an easily calculated fitness function, the TSP is well suited to genetic algorithms.

GAs are optimization algorithms that work with a population of individuals and they are based on the Darwinian theory of natural selection and evolution. Firstly, an initial population of individuals is created, which is often accomplished by a random sampling of possible solutions. Then, each solution is evaluated to measure its fitness. After that, variation operators (such as mutation and crossover) are used in order to generate a new set of individuals. A mutation operator creates new individuals by performing some changes in a single individual, while the crossover operator creates new individuals (offspring) by combining parts of two or more other individuals (parents) [1]. And finally, a selection is performed, where the most fit individuals survive and form the next generation. This process is repeated until some termination condition is reached, such as obtaining a good enough solution, or exceeding the maximum number of generations allowed.

In our case a tour is represented as a path, in which a city at the position i is visited after the city at the position $i-1$ and before the city at the position $i+1$. For the TSP the evaluation part of the algorithm is very straightforward, i.e. the fitness function of a tour corresponds to its length. The mutation operator randomly picks two cities in a tour and reverses the order of the cities between them (i.e. the mutation operator tries to repair a tour that crosses its own path). We used a partially-mapped (PMX) crossover proposed in [10], which produces an offspring by choosing a subsequence of a tour from one parent and preserving the order and position of as many cities as possible from the other parent. A subsequence of a tour that passes from a parent to a child is selected by picking two random cut points. So, firstly the segments between the cut points are copied from the parent 1 to the offspring 2 and from the parent 2 to the offspring 1. These segments also define a series of mappings. Then all the cities before the first cut point and after the second cut point are copied from the parent *1* to the offspring *1* and from the parent *2* to the offspring *2*. However, this operation might result in an invalid tour, for example, an offspring can get duplicate cities. In

order to overcome this situation, a previously defined series of mappings is utilized, that indicate how to swap conflicting cities.

For example, given two parents with cut points marked by vertical lines:

$$p_1=(3 \mid 0\ 1\ 2 \mid 4)$$

$$p_2=(1 \mid 0\ 2\ 4 \mid 3)$$

the PMX operator will define the series of mappings ($0\leftrightarrow0$, $1\leftrightarrow2$, $2\leftrightarrow4$) and produce the following offspring:

$$o_1=(3 \mid 0\ 2\ 4 \mid 1)$$

$$o_2=(4 \mid 0\ 1\ 2 \mid 3)$$

In order to choose parents for producing the offspring, a fitness proportional selection is employed. For this we use a roulette wheel approach, in which each individual is assigned a slot whose width is proportional to the fitness of that individual, and the wheel is spun each time a parent is needed. We apply also an elitist selection, which guarantees the survival of the best solution found so far.

The algorithm described was implemented in a software application developed in C++ language. After that, a number of experiments have been conducted with benchmarks from the TSPLIB [11]. The experiments were performed with different crossover rates (10%, 25% and 50%), and they have shown that a significant percentage of the total execution time is spent performing the crossover operation (it ranges from 15% to 60%). That is why we implement firstly the crossover operation in FPGA in order to estimate an efficiency of such an approach. The results of some experiments are presented in Table 1.

Table 1. The results of experiments in software

Name	Crossover rate – 25%			Crossover rate – 50%		
	t_{total} (s)	t_{cros} (s)	$\%_{cros}$	t_{total} (s)	t_{cros} (s)	$\%_{cros}$
a280	5.88	1.49	25.3	8.39	3.66	43.6
berlin52	1.27	0.32	25.2	1.79	0.75	41.9
bier127	2.75	0.68	24.7	3.92	1.68	42.9
d657	15.18	4.62	30.4	23.33	11.92	51.1
eil51	1.22	0.31	25.4	1.76	0.74	39.2
fl417	9.30	2.62	28.2	13.56	6.36	46.9
rat575	12.97	3.78	29.1	19.46	9.53	48.9
u724	17.02	5.36	31.5	25.6	13.19	51.5
vm1084	28.09	9.69	34.5	43.63	24.13	55.3

The first column contains the problem name (the number included as part of the name shows the corresponding number of cities). The columns t_{total} store the total execution time in seconds on a PentiumIII/800MHz/256MB running Windows2000. For all the instances the population size was of 20 individuals, and 1000 generations were performed. The columns t_{cros} record the time spent performing the crossover operation. And finally, the columns $\%_{cros}$ indicate the ratio of the crossover operation comparing to the total execution time (in %).

3 Hardware Implementation of the Crossover Operation

The suggested architecture of the respective circuit is depicted in Fig. 2. It includes a central control unit, which activates in the required sequence all the steps of the algorithm needed to be performed. The current version of the architecture supports tours composed of at most 1024 cities. Thus there are four memories of size 2^{10}x10 that are used in order to keep two parent tours and two resulting offspring. A city at the memory address *i* is visited after the city at the address *i-1* and before the city at the address *i+1*. Actually, such a large number of cities (i.e. destinations) is not necessary for the distribution system considered. The required maximum number can be much less (we think 64 is enough).

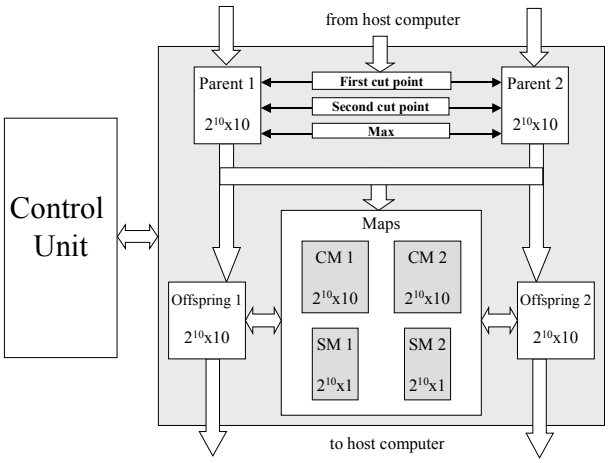

Fig. 2. The proposed architecture for realizing the crossover operation

The cut points used in a crossover are randomly chosen by the software application and stored in two special 10-bit registers ("First cut point" and "Second cut point" in Fig. 2). The third 10-bit register ("Max" in Fig. 2) stores the actual length of a tour. Taking into account this value, the control unit will only force processing of the required area of parents' and offspring memories. It allows accelerating the crossover for tours of small dimensions.

Four additional memories are utilized to help in performing the crossover operation. These memories are "CM1" and "CM2" of size 2^{10}x10 (we will refer to them as complex maps) and "SM1" and "SM2" of size 2^{10}x1 (we will refer to them as simple maps).

In order to perform the PMX crossover, the following sequence of operations has to be realized. Firstly, the values of cut points and the length of a tour for a given problem instance are downloaded to the FPGA. Then, the parent tours are transferred from the host computer to the memories "Parent 1" and "Parent 2". Each time a city is written to parent memories, the value "0" is also written to the same address in the respective simple map (actually, it allows the simple maps to be reset). After that the segments between the cut points are swapped from the "Parent 1" to the "Offspring 2"

and from the “Parent 2” to the “Offspring 1”. Each time we transfer a city *c1* from the “Parent 1” to the “Offspring 2”, a value “1” is written to the simple map “SM2” at the address *c1*. The same thing occurs with the second parent, i.e. when we transfer a city *c2* from the “Parent 2” to the “Offspring 1”, a value “1” is written to the simple map “SM1” at the address *c2* as well. At the same time, the value *c1* is stored at the address *c2* in the complex map “CM1”, and the value *c2* is stored at the address *c1* in the complex map “CM2”. At the next step, all the cities before the first cut point, and after the second cut point should be copied from the “Parent 1” to the “Offspring 1” and from the “Parent 2” to the “Offspring 2”, and any conflicting situations must be resolved.

For this the following strategy is utilized (see Fig. 3). Firstly, a city *c* is read from the “Parent 1”. If a value “0” is stored in the simple map “SM1” at the address *c*, then this city can be safely copied to the “Offspring 1”. In the opposite case, if value “1” is stored in the simple map “SM1” at the address *c*, it means that this city has already been included into the “Offspring 1”. Consequently, it should be replaced with some other city. For this purpose the complex map “CM1” is employed as shown in the flow-chart in Fig. 3. The same operations are also performed to fill the second offspring, the only difference being that maps “SM2” and “CM2” are employed. And finally, the offspring are transferred to the host computer.

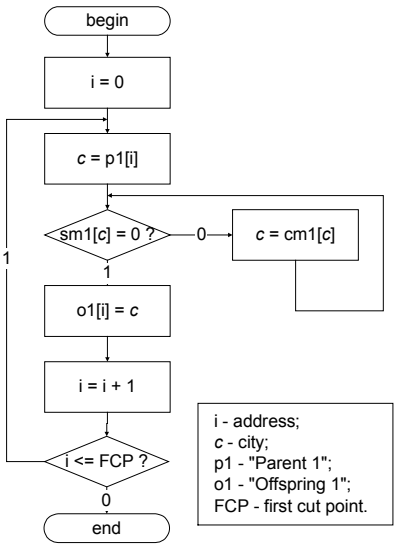

Fig. 3. Filling of the “Offspring 1” before the first cut point

We used an ADM-XRC PCI board [12] containing one XCV812E Virtex Extended Memory FPGA [13] as the hardware platform. This type of FPGA is very well suited to the proposed architecture because it incorporates large amounts of distributed RAM-blocks, which can be used to store parent and offspring tours and maps. The remaining three major parts of the GA, which are evaluation, mutation and selection, are not implemented yet. That is why currently they are performed in a software application communicating with the FPGA with the aid of the ADM-XRC API

library, which provides support for initialization, loading configuration bit streams, data transfers, interrupts processing, clock management and error handling.

Table 2 contains information about the area occupied by the circuit implementation and its clock frequency. The area is shown in the number of Virtex slices (each configurable logic block (CLB) is composed of two slices). In the parentheses it is shown how much resources (in %) of XCV812E the circuit consumes.

Table 2. Parameters of the circuit implemented

Area (*slices*)	**Number of RAM-blocks**	**Maximum clock frequency (MHz)**
149 (1%)	20 (7%)	102.659

4 Hierarchical Specification of Control Algorithms

There are two methods that are most appropriate for the specification of hierarchical control algorithms. The first method is called Statecharts [14] and the second is called hierarchical graph-schemes (HGS) [3]. We employ the second method because it permits a synthesizable specification to be built and supports virtualization of control algorithms.

It is obvious that the flow of sub-algorithms can be obtained directly from the result of solving the respective TSP problem. For the considered technique this flow includes only unconditional transitions (see Fig. 4).

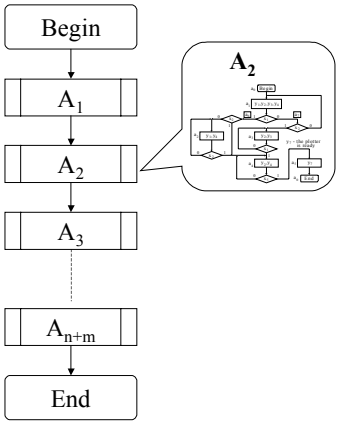

Fig. 4. The required sequence of sub-algorithms and the resulting specification in form of HGS

We assume that all the sub-algorithms are known, and the respective specifications are stored in a library. However, initially all these library sub-algorithms are unlinked. Thus the objective is to construct the main algorithm that invokes the respective sub-algorithms in the required order (see Fig. 4). In fact this task is trivial. The problem might appear when we want to synthesize a run-time reprogrammable control circuit

that implements the required sequence of sub-algorithms. This is because the number of sub-algorithms and the contents of each rectangular node (see Fig. 4) are unknown before execution time. Besides, during run-time the main algorithm might need to be reconstructed. As a result the respective control circuit has to be virtual.

More complicated practical problems might also require conditional transitions between sub-algorithms. For example, such transitions will appear if there is more than one optimum result for the step 1 (for example, there are two shortest ways of the same length). Thus, in general case the main algorithm can include not only unconditional but also conditional transitions.

5 Synthesis and Implementation of Virtual Control Circuits

The property of virtualization states that the desired control algorithm has to be implemented by modifying the functionality of the existing hardware dynamically. This can be achieved with the aid of so-called hardware templates (HT). A HT includes modifiable components (such as RAM or partially dynamically reconfigurable areas of FPGA) connected to each other in a certain way. One possible architecture of HT, modeled by a RAM-based finite state machine (FSM), was considered in [4]. The HT has predefined constraints, such as the maximum number of inputs (they affect conditional transitions), the maximum number of activated sub-algorithms, etc. However, all these constraints can easily be estimated from the initial specification of the problem. In [4] it is shown that reloading the respective RAM-blocks can provide all changes in the functionality of the control circuit. This technique gives a complete solution for the virtual control circuit, making the main algorithm synthesizable during execution time.

The next problem to deal with is that the links between the rectangular nodes of the main algorithm and the activating sub-algorithms should be dynamically modifiable. In order to resolve this situation, the FSM memory has to be based on a stack. From the beginning the value on the top of the stack points to the main algorithm and the method [4] can be used to fill the respective RAM-blocks. Invocation of any sub-algorithm increments the stack pointer and a new value at the stack output indicates the respective sub-algorithm. Since all the autonomous sub-algorithms are known before execution time, the respective data for RAM-blocks can be obtained in advance. The sub-algorithms can be swapped either by changing entry points for preliminary loaded RAM-blocks, or by run-time reloading of RAM-blocks. Since the main algorithm is basically composed of sequential chains, we can combine an execution of some sub-algorithm with the reloading of RAM-blocks for the next sub-algorithm.

Finally, the process of synthesis and implementation includes the following steps.

1. Translate the main algorithm into a set of bit streams that have to be loaded into modifiable RAM-blocks of HT. This permits a skeletal frame of the main algorithm to be constructed.
2. Establish links between the rectangular nodes of the skeletal frame and real sub-algorithms that have to be implemented.
3. Load the RAM-blocks of the FPGA.

6 Experiments

A comparison between software and hardware versions of the crossover operation is presented in Table 3. The first column in Table 3 stores the number of cities in a tour. The column t_{soft} contains the time of performing the crossover in software. The software version was based on the C++ language and executed on a PentiumIII/800MHz/256MB running Windows2000. The column t_{hard} records the time for performing the crossover in hardware. The hardware part was executed on an XCV812E FPGA with a clock frequency of 40 MHz. The cut points chosen in software and hardware versions were the same in order to facilitate the comparison. The speedup resulting from our approach is given by t_{soft}/t_{hard}. The results of experiments show that the PMX crossover operator is executed in FPGA 10-50 times faster than in software.

Table 3. Comparison of software and hardware implementations of the PMX crossover operator

Number of cities	t_{soft} (ms)	t_{hard} (ms)	Speedup
280	1.0896	0.08443	12.9
52	0.4293	0.01872	22.9
127	0.6203	0.04040	15.4
657	3.838	0.19486	19.7
417	2.0605	0.12524	16.5
442	3.8394	0.13045	29.4
575	2.7703	0.17199	16.1
724	10.2145	0.20443	49.9

It should be noted that the time t_{hard} (and, consequently, the speedup achieved) strongly depends on the cut points chosen. This is because the first part of the PMX crossover (swapping the segments between the cut points) is executed in parallel for both parent-offspring pairs. Moreover, this part involves just simple reading from parent memory and sequential writing to offspring memory without performing any other operation. As a result, it concludes very quickly. On the other hand the second part of the crossover (filling offspring before the first and after the second cut points) is performed sequentially for each parent-offspring pair. Besides of executing memory reading/writing operations, some checks are also performed, that ensure only valid tours are constructed. As a result, the greater the distance between the cut points, the faster the PMX crossover will be completed in FPGA.

The acceleration achieved in FPGA compared to a software implementation can be explained by the following primary reasons. Firstly, the technique of parallel processing is employed, i.e. a part of the crossover operation is executed in parallel in FPGA. Second, the memory organization in the proposed architecture is tailored to the required data sizes. We think it is possible to provide further acceleration in the FPGA by implementing the second part of the PMX crossover also in parallel. In order to do this just simple changes in the control unit are required.

7 Conclusion

The paper presents a solution for synthesis and implementation of a reconfigurable adaptive control system, which can be used for manufacturing applications. The problem considered has been informally specified and converted to the TSP. The latter was solved with the aid of a GA. At the current implementation version, the most computationally intensive task (the crossover) is assigned to hardware, while the other tasks are performed in software. The result of solving the TSP permits the construction of a skeletal frame for future specification of the control circuit. Since the circuit is virtual, it is constructed without any knowledge of its functionality (in fact only some predefined constraints have been taken into account). It is structurally organized as a hardware template. A particular functionality can be implemented during execution time by reloading/configuring dynamically modifiable blocks. We believe that the results of the paper have shown one of the first engineering examples that demonstrates the practical implementation of virtual control circuits by combining methods and algorithms from different scientific areas, such as a modern methodology in logic synthesis and an evolutionary approach in artificial intelligence.

References

1. Michalewicz, Z., Fogel, D.B.: How to Solve It: Modern Heuristics. Springer (2000)
2. Sklyarov, V., Skliarova, I., Ferrari, A.B.: Hierarchical Specification and Implementation of Combinatorial Algorithms Based on RHS Model. Proc. of the XVI Conference on Design of Circuits and Integrated Systems – DCIS'2001, Portugal (2001) 486-491
3. Sklyarov, V.: Hierarchical Finite-State Machines and Their Use for Digital Control. IEEE Trans. on VLSI Systems, Vol. 7, No 2 (1999) 222-228
4. Skliarova, I., Ferrari, A.B.: Synthesis of reprogrammable control unit for combinatorial processor. Proc. of the 4th IEEE Int. Workshop on Design and Diagnostics of Electronic Circuits and Systems – IEEE DDECS'2001, Hungary (2001) 179-186
5. Golden, B.L., Kaku, B.K.: Difficult Routing and Assignment Problems. In: Rosen, K.H., Michaels, J.G., Gross, J.L., Grossman, J.W., Shier, D.R. (eds.): Handbook of Discrete and Combinatorial Mathematics. CRC Press, 692-705 (2000)
6. Junger, M., Reinelt, G., Rinaldi, G.: The traveling salesman problem. In: Ball, M., Magnanti, T., Monma, C., Nemhauser, G. (eds.): Network Models, 225-330 (1995)
7. Litke, J.D.: An Improved Solution to the Traveling Salesman Problem with Thousands of Nodes. Communications of the ACM, vol. 27, No. 12 (1984) 1227-1236
8. Ahuja, R.K., Magnanti, T.L., Orlin, J.B., Reddy, M.R.: Applications of network optimization. In: Ball, M., Magnanti, T., Monma, C., Nemhauser, G. (eds.): Network Models, 1-83 (1995)
9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-completeness. Freeman (1979)
10. Goldberg, D.E., Lingle, R.: Alleles, Loci, and the Traveling Salesman Problem. Proc. of the 1st Int. Conf. on Genetic Algorithms, 154-159 (1985)
11. http://www.informatik.uni-heidelberg.de/groups/comopt/software/ TSPLIB95/index.html
12. http://www.alphadata.co.uk
13. Xilinx: The programmable logic data book. Xilinx, San Jose (2000)
14. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, N8 (1987) 231-271