A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2

Paul Graham and Brent Nelson*

Reconfigurable Logic Laboratory
Brigham Young University, Provo UT 84602, USA
801-378-4012
grahamp@fpga.ee.byu.edu
nelson@ee.byu.edu

Abstract. With the introduction of Splash, Splash 2, PAM, and other reconfigurable computers, a wide variety of algorithms can now be feasibly constructed in hardware. In this paper, we describe the Splash 2 Parallel Genetic Algorithm (SPGA), which is a parallel genetic algorithm for optimizing symmetric traveling salesman problems (TSPs) using Splash 2. Each processor in SPGA consists of four Field Programmable Gate Arrays (FPGAs) and associated memories and was found to perform 6.8 to 10.6 times the speed of equivalent software on a state-of-the-art workstation. Multiple processor SPGA systems, which use up to eight processors, find good TSP solutions much more quickly than single processor and software-based implementations of the genetic algorithm. The four-processor island-parallel SPGA implementation out performed all other SPGA configurations tested. We conclude noting that the described parallel genetic algorithm appears to be a good match for reconfigurable computing machines and that Splash 2's various interconnect resources and support for linear systolic and MIMD computing models was important for the implementation of SPGA.

1 Introduction

The development of reconfigurable computers such as Splash [1], Splash 2 [2], and PAM [3] makes it possible to quickly create custom hardware implementations of a wide range of algorithms. In some cases, classes of algorithms not previously suitable for ASIC realization can now benefit from hardware implementation on these machines. Splash 2 and PAM applications have been shown to often run orders of magnitude faster than equivalent software implementations[4][5]. In this work we describe the result of mapping a genetic algorithm for solving the traveling salesman problem onto the Splash 2 system. We demonstrate the

* This work was supported by ARPA/CSTO under contract number DABT63-94-C-0085 under a subcontract to National Semiconductor

effectiveness of Splash 2 for this and the performance advantages of using a reconfigurable computer for this computation.

We begin with a brief overview of genetic algorithms, providing a description of our specific algorithm for the traveling salesman problem. We then describe a basic implementation using four FPGAs and compare its speed to an equivalent software implementation. Two parallel versions of the algorithm, taking full advantage of the parallel capabilities of Splash 2, are then presented along with results and conclusions.

2 Genetic Algorithms

In 1975, Holland [6] developed an optimization technique, based on the process of natural selection and evolution, which he called the *genetic algorithm* (GA). Follow-on work since that time has shown its usefulness for optimization problems requiring the search of large and complex problem spaces from engineering design to combinatorial optimization to control [7][8][9].

During its operation the GA maintains a collection (*population*) of candidate solutions. Associated with each candidate is a *fitness* or measure of its quality. The algorithm proceeds by selecting candidates from the current generation to propagate into the next generation. In the process of this propagation the algorithm may simply copy the selected candidates to the new generation or it may combine pairs of candidates through a *crossover* operation, reminiscent of mating in natural systems. In this case, the newly created solutions have characteristics taken from both parents. The selection of candidates for copy and crossover is randomized but biased toward candidates with higher fitnesses, thus, more fit individuals are more likely to be used to produce future generations of solutions. As a means of preventing premature convergence to local minima, an operation known as *mutation* randomly perturbs solutions to yield new ones not otherwise related to existing solutions.

2.1 A Genetic Algorithm for the Traveling Salesman Problem

SPGA (Splash 2 Parallel Genetic Algorithm) is a hardware GA which searches for optimal solutions to symmetric traveling salesman problems (TSPs). This family of problems involves finding the shortest path through a collection of n cities, visiting each city exactly once and returning to the starting city. Software implementations of this algorithm have previously been studied—this is the first known hardware implementation of a GA for this problem.

Each candidate solution in the population consists of an ordered list describing the sequence in which each city is visited—a list referred to as a *tour*. Since the object of optimizing a TSP is to find the shortest tour, the fitness of a given tour is related to its length.

Crossover is performed in the following way: once a pair of tours (call them tours A and B) have been selected for crossover a cut point is selected at random and the two tours are cut at that point. The head of tour A is used as the head of offspring #1 and the head of tour B is used as the head of offspring #2. The tail of offspring #1 is formed by taking, in order, the cities from tour B not contained in the head of tour A. The tail of offspring #2 is formed in a similar fashion. Mutation is performed on selected tours by reversing the order of cities visited within a sub-tour contained within the original tour. The endpoints of this sub-tour are chosen randomly for each tour mutated. These crossover and mutation operations closely follow those described in [10]. The reader is directed there for details on their operation.

The above algorithm readily lends itself to parallelization in at least two ways. The brute force approach is to run multiple independent copies of the algorithm, selecting the best result produced by the collection of runs. In contrast, a cooperative parallel model can be employed where *islands* of computation are executed in parallel but periodically exchange solutions with one another through *migration*. This often seeds searches into new and better areas of the search space and has been shown to be robust in dealing with a variety of difficult optimization problems due to its tendency to avoid premature convergence to local minima[11].

3 Splash 2

SPGA was developed on Splash 2, a reconfigurable computer developed at the Center for Computing Sciences (formerly the Supercomputing Research Center). Splash 2 was designed to support linear systolic, SIMD, and MIMD processing styles. Splash 2 is hosted on a Sun Sparc and consists of an interface board and a collection of processor array boards as shown in Figure 1. The interface board provides support for the host to program, control, and exchange data with the Splash 2 array boards.

Splash 2 is "programmed" using VHDL[12]. Commercial synthesis tools are used to map the VHDL design onto the FPGA resources of Splash 2, while auxiliary text files provide configuration information for the memories and crossbars. VHDL models for the entire Splash 2 system including the interface board, buses, crossbars, and local processor memories are available and provide a complete and accurate pre-synthesis simulation environment.

4 Basic Algorithm Implementation

SPGA's basic unit of computation is a *processor*, consisting of four FPGAs (Xilinx 4010s) and their associated memories. The FPGAs are arranged in a bidirectional pipeline, as shown in Figure 2. During execution the memories hold

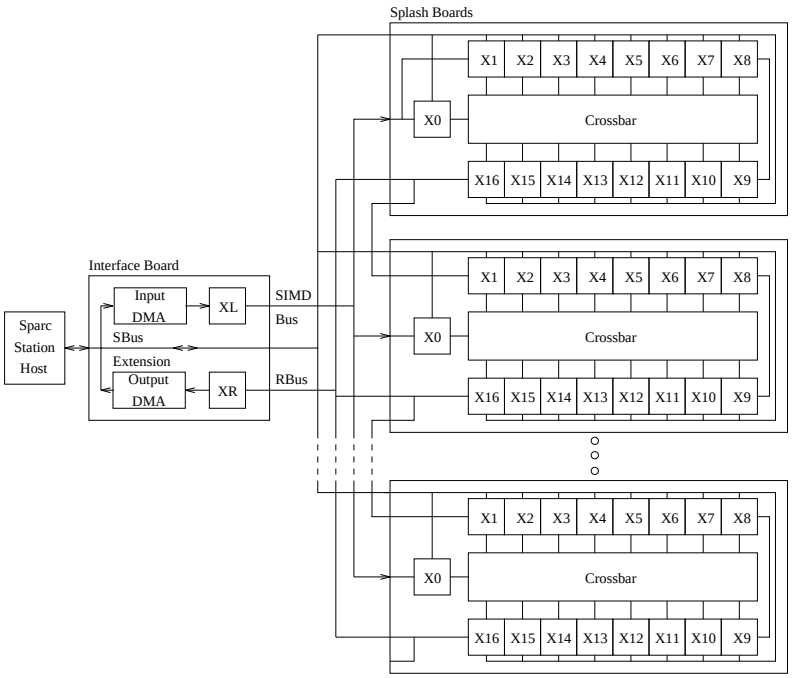

Fig. 1. The Splash 2 System

the current generation, the new generation as it is constructed, tour fitnesses, the inter-city distance matrix, and operational parameters. Initial population data is provided by the host driver program.

The function of the FPGAs is as follows:

- *FPGA 1* performs the biased selection, choosing tour pairs from memory. This is done using a hardware-pipelined *roulette wheel* algorithm. Initially, a random number is generated called the *target*. Tour fitness values are then sequentially accumulated until the target value is exceeded. The tour causing the overflow is the next tour selected. The width of each tour's roulette wheel slot is proportional to its fitness, therefore, highly fit tours are more likely to be chosen. Once a pair of tours is selected, their index numbers are passed to *FPGA 2* via the bottom pipeline path.

- *FPGA 2* has two choices when given a tour pair: it can simply copy the tours to the right unchanged or it can combine them via crossover and send the new offspring to the right. This decision is made based on a random number generated on chip. Crossover probabilities of 10% to 60%

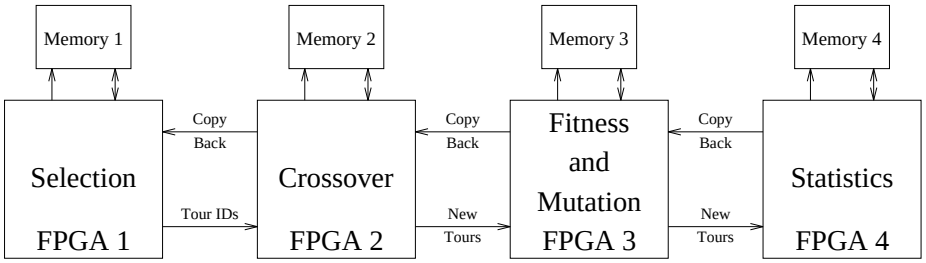

Fig. 2. A Single Genetic Algorithm Processor

have been shown to give good results for TSPs.

- The role of *FPGA 3* is threefold. First, it computes new fitness values for tours formed by crossover. Second, it randomly selects tours for mutation and performs the mutation. Finally, it sends the tour pairs and their fitnesses to *FPGA 4*.
- *FPGA 4* writes the new population to memory, determining and recording the best and worst tours for this generation and the best tour to date.

The above process is repeated until a population equal in size to the original population is collected in the memory of *FPGA 4*. The upper pipeline path in Figure 2 is then used to copy the new population and fitnesses back to the memories of *FPGA 1* and *FPGA 2*. This ends the computation of a complete generation. Execution terminates when the desired number of generations has been reached.

The implementation required 3,500 lines of VHDL code. The maximum clock frequency of 11 MHz was dictated by the arithmetic sections of FPGA's 1 and 4. CLB utilizations for the four chips range from 37% to 60%. Also, a family of C-based driver programs were written for the SPARC host to control the computation and retrieve the results.

The performance of a software implementation executing on a 125 MHz HP PA-RISC workstation is compared to SPGA in Table 1. All numbers are averages across a collection of runs. The 24 city problems ran for 120 million cycles, completing 2,000 generations, while the 120 city problems ran for 3.4 billion cycles, completing 20,000 generations. The software and hardware implementations found similar quality solutions in our tests.

The significant advantage of the FPGA design over software was initially surprising, given the 11x difference in clock rates. We attribute the modest software performance to operating system overhead, TLB and cache misses, complex addressing calculations, and strict sequential thread of control. However, we have little quantitative information on these effects. In contrast, SPGA employs a

Num. of Cities	*Pop. Size*	*Crossover Probability (%)*	*Mutation Probability (%)*	*Ave. Exec. Time (sec)*		*Soft./ Hard*
				Hardware	*Software*	
24	128	10	10	4.38	43.7	9.97
24	256	10	10	11.23	118.7	10.57
120	256	60	10	295	1999.9	6.78

Table 1. Comparison of Hardware and Software Execution Times

custom 4-stage pipeline, achieves nearly 100% memory bandwidth utilization, and incurs no overhead for the operating system, address calculations, or cache misses.

5 Two Parallel Implementations

The implementation described above uses only 4 FPGAs on a single Splash 2 array board—the remaining 30 FPGAs and memories in our two-board Splash 2 are idle. In fact, there was no need for Splash 2 at all for the basic implementation. Four FPGAs and memories would have been sufficient. However, given a two-board Splash 2, it is a simple extension to run seven additional copies of the algorithm—no additional hardware design is required. Since the copies of the algorithm do not interact, the only added overhead is the small amount of time spent initializing the additional memories. Thus, an eight-fold increase in search rate is possible with this approach which we call the *trivially parallel* model.

A more interesting approach to parallelizing the algorithm applies a cooperative or *island* model of parallel computation. In this model several searches are conducted simultaneously with periodic migration of solutions between search islands. Figure 3 gives a diagram of a single array board design for this model.

During migration each island, in turn, broadcasts a subset of its tours to the other islands via the crossbar. The receiving islands replace random solutions in their own populations with these broadcast tours. Unlike the trivially parallel model described above, modifications to the original SPGA design were required for this. First, the islands employ dedicated lines in Splash 2 to signal to X0 when they are ready to perform migration. Second, X0 coordinates these migration transfers by configuring the crossbar and handshaking with each island in turn as required. This added functionality required the creation of an X0 design, significant additions to FPGA 4's design, and the creation of multiple crossbar configurations.

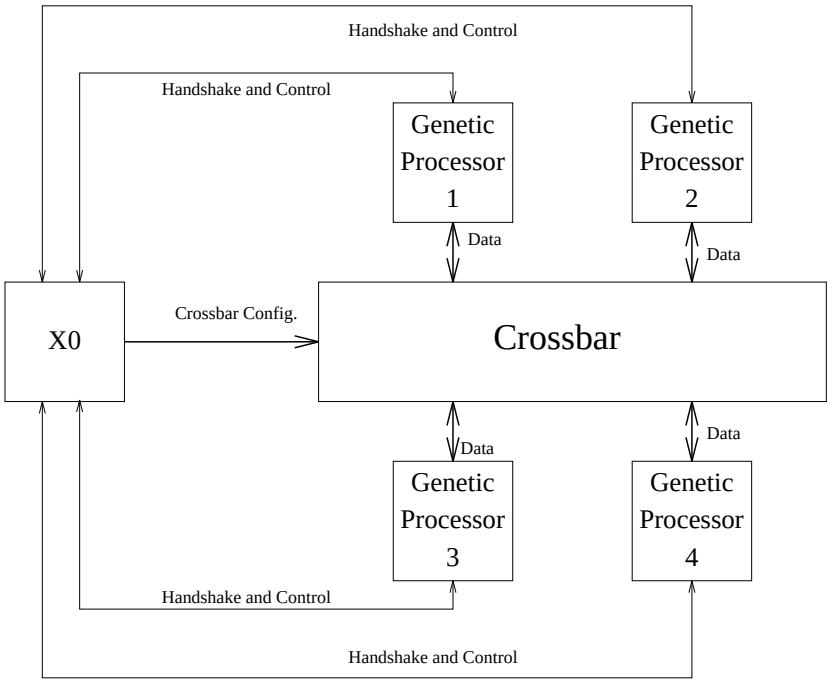

Fig. 3. The Island Parallel Model

5.1 Performance Results

Figure 4 compares the Splash 2 performance of the three computation models described above. These include a 1-processor design, 4-processor and 8-processor trivially parallel designs, and a 4-processor island model. All results are averages across many executions.

To a first order approximation, the rate of search is proportional to the number of search engines. Thus, using the numbers from Table 1 the 8-processor trivially parallel version would be expected to search faster than the software by factors from 54 to 85. However, these speedups do not translate directly to better solutions since the quality of solution obtained isn't linearly proportional to the number of solutions evaluated.

If one's goal is to find the best solution possible we have found that more processors are always better, but marginally so. For long runs (3.5 billion cycles) the 8-processor parallel version gives an answer about 4% better than a single processor and the 4-processor island model is better by about 6%.

However, if the goal is to find a good solution quickly, the data shows that the island model is far superior to the others. At 500 million cycles the 4-

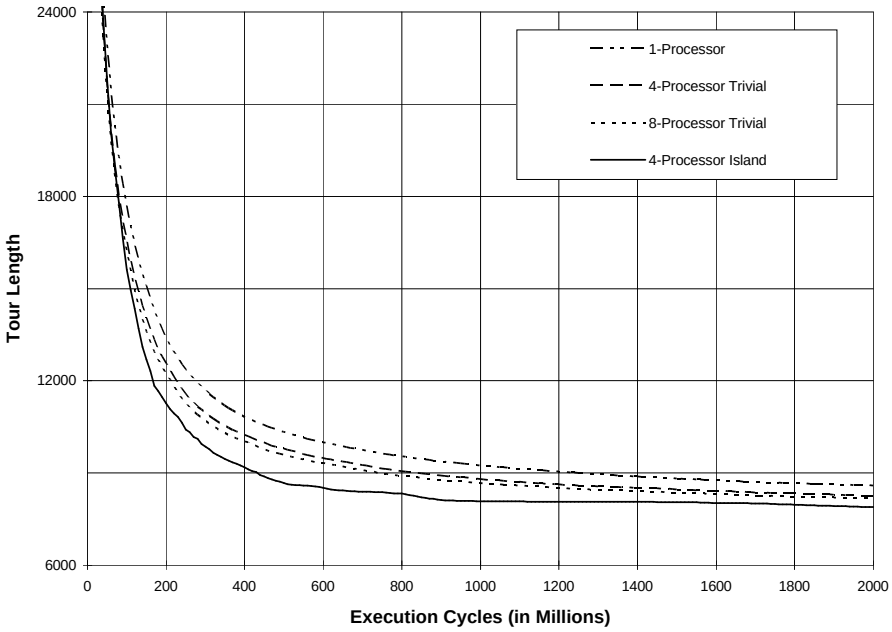

Fig. 4. GA Performance Comparison

processor island model has found a solution which requires 990 million cycles for the 8-processor trivially parallel model to match and 1.7 billion cycles for the single-processor to match.

6 Conclusions

We have described a hardware genetic algorithm on Splash 2. We believe the genetic algorithm described in this paper is a good match for reconfigurable computers for a number of reasons.

- We have shown that reasonably modest hardware resources (4 FPGAs and 4 memories) can significantly outperform state-of-the-art workstations on this algorithm. SPGA does this with a custom design that avoids operating system overhead, TLB and cache misses, and complex addressing calculations. In addition, it makes use of pipelining to achieve parallel execution.

- The individual data objects manipulated by the algorithm are small, mostly 8- or 16-bits with a few 24-bit values. This is a a good match for the computation and storage capabilities of today's FPGAs.
- The arithmetic requirements of SPGA are modest, consisting of small-word additions, subtractions, and comparisons. This is also a good match for today's FPGAs.
- The additional work required to create parallel implementations of the algorithm is minimal. The parallel versions were able to find 'good' solutions in far less time than the single processor version.
- Other search methods often terminate at a specific solution; genetic algorithm search methods continue to find better solutions until terminated by the user. Custom hardware such as described herein may be required to take advantage of this for very long execution times.

A further conclusion of this project is that since few entire computations are purely systolic or SIMD, extra interconnections are often required for a complete implementation and may make the difference between poor and excellent performance. The breadth of computing models and FPGA interconnection paths supported by the Splash 2 proved most useful for SPGA. The linear pipeline was required for each processor's design and additionally the MIMD model was essential for supporting migration in the island-parallel version. All of the Splash 2 interconnect paths were eventually used in the design including: nearest-neighbor linear connections, the crossbar, and the X0 broadcast and handshake lines.

Areas for future work with SPGA include porting designs similar to SPGA to other reconfigurable hardware platforms, experimenting with different migration strategies, extending the system to allow migration between array boards, and using SPGA as a test bed for furthering parallel genetic algorithm research.

References

1. M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, January 1991.
2. J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.
3. P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirther, and E. Swartslander Jr., editors, *Systolic Array Processors*, pages 300–309. Prentice Hall, 1989.
4. D. P. Lopresti. Rapid implementation of a genetic sequence comparator using field-programmable gate arrays. In C. Sequin, editor, *Advanced Research in VLSI:*

Proceedings of the 1991 University of California/Santa Cruz Conference, pages 138–152, Santa Cruz, CA, March 1991.

5. P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.
6. J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.
7. R. Vemuri, R.; Vemuri. Mcm layer assignment using genetic search. *Electronics Letters*, 30(20):1635–7, September 1994.
8. N.; Hong Ren Mou, E.S.H.; Ansari. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed System*, 5(2):113–120, February 1994.
9. John E. Lansberry and L. Wozniak. Adaptive hydrogenerator governor tuning with a genetic algorithm. *IEEE Transactions on Energy Conversion*, 9(1):179–183, March 1994.
10. M. C. Leu; H. Wong; and Z. Ji. Planning of component placement/insertion sequence and feeder setup in pcb assembly using genetic algorithm. *Transactions of the ASME*, 115:424–432, December 1993.
11. M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In J. Stender, editor, *Parallel Genetic Algorithms: Theory and Applications*, pages 5–41. IO Press, Washington DC, 1993.
12. J. M. Arnold, D. A. Buell, and E. G. Davis. VHDL programming on Splash 2. In *More FPGAs: Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications*, pages 182–191, Oxford, England, September 1993.