A Fast FPGA-Based 2-Opt Solver for Small-Scale Euclidean Traveling Salesman Problem

Ioannis Mavroidis, Ioannis Papaefstathiou, Dionisios Pnevmatikatos
Microprocessor and Hardware Lab (MHL)
Technical University of Crete (TUC)
Kounoupidiana, Crete, GR73100, Greece
{maurog, ygp, pnevmati}@mhl.tuc.gr

Abstract

In this paper we discuss and analyze the FPGA-based implementation of an algorithm for the Traveling Salesman Problem (TSP), and in particular of 2-Opt, one of the most famous local optimization algorithms, for Euclidean TSP instances up to a few hundred cities. We introduce the notion of "symmetrical 2-Opt moves" which allows us to uncover fine-grain parallelism when executing the specified algorithm. We propose a novel architecture that exploits this parallelism, and demonstrate its implementation in reconfigurable hardware. We evaluate our proposed architecture and its implementation on a state-of-the-art FPGA using a subset of the TSPLIB benchmark, and find that our approach exhibits better quality of final results and an average speedup of 600% when compared with the state-of-the-art software implementation. Our approach produces, to the best of our knowledge, the fastest to date TSP 2-Opt solver for small-scale Euclidean TSP instances.

1 Introduction

The Traveling Salesman Problem (TSP) is the problem of a salesman who starts from his hometown and wants to visit a specified set of cities, returning to his hometown at the end. Each city has to be visited exactly once and the requirement is to find the shortest possible tour. Stated more formally, the TSP seeks the shortest Hamiltonian cycle in a weighted graph whose vertices correspond to cities and edge weights correspond to distances between cities. In this paper we will concentrate on the symmetric TSP, in which, going from city A to city B has the same distance/weight as going from city B to city A. More specifically, we will concentrate on very widely used fully-connected Euclidean instances of the TSP, where each city is represented by its two coordinates.

The symmetric TSP is NP-hard and is one of the best-known and well-studied combinatorial optimization problems. It has many practical applications ranging from CAD tools for VLSI chip implementation to DNA mapping and X-ray crystallography.

Since finding the tour with the minimum length is an NP-hard problem, most algorithms concentrate on finding near-optimal tours as quickly as possible. Local search algorithms start from an initial ordering of the cities and attempt to improve this ordering by performing simple tour modifications. Each such algorithm has a specified set of allowed tour operations (or moves) that it can use to convert one tour into another, and will repeatedly perform these operations, as long as each reduces the length of the current tour, until no further improvement can be made (i.e. a locally optimal tour has been reached). However, a locally optimal tour may not necessarily be close to the globally optimal tour. In order to escape from local minima, we may want to modify this basic scheme of pure optimization and also allow "uphill" moves in our search for the global minimum. Simulated annealing [6] is a well-known algorithm that does just that. It allows "uphill" moves based on a carefully crafted probability function.

One efficient algorithm to find TSP tours is the 2-Opt algorithm, described in detail in Section 3. In this paper we present an architecture that exploits fine-grain parallelism in evaluating and applying 2-Opt moves during the search for a locally optimal tour. The architecture is evaluated in terms of required hardware, quality of results and speed. The main contribution of this paper is that it is one of the very few implementations in reconfigurable hardware (and in hardware in general) of a TSP solver. Moreover, this is to the best of our knowledge, the most efficient TSP solver for TSP instances up to a few hundred cities using the 2-Opt algorithm, since it is on average 600% faster than the fastest software approach, while it also produces better quality (i.e. closer to the optimal) results.

An earlier and shorter paper [13] describes the main points of the proposed algorithm and the feasibility of this approach for fixed hardware. The new material included in this work can be mainly summarized in the following:

1. We present a new scalable architecture tailored to reconfigurable hardware, while we describe in detail an updated and more effective version of our algorithm. Our new proposed architecture allows the number of cities to scale independently of the number of Processing Elements (PEs).
2. We discuss the deterministic nature of our approach and evaluate it against a widely-used randomized 2-Opt algorithm.
3. We provide detailed implementation results in terms of performance and cost for a state-of-the-art FPGA.

The remainder of this paper is organized as follows. Section 2 shows some related work, and Section 3 provides the necessary background on 2-Opt. Section 4 introduces the notion of "*symmetrical 2-Opt moves*" and discusses their benefits. Section 5 describes the proposed architecture and Section 6 shows some implementation results for a Xilinx Virtex II Pro FPGA. Section 7 investigates how the deterministic nature of the architecture affects its performance, and Section 8 reports detailed performance results and comparisons against the Concorde software package. Finally, in Section 9 we present our ongoing and future work and in Section 10 we conclude.

2 Related Work

The TSP is probably the most-studied optimization problem of all time. Many researchers, both mathematicians and computer scientists, have attacked this problem for decades resulting in a plethora of heuristics that offer a broad range of tradeoffs between running time and quality of solution. The heuristics range from tour construction, local optimization, and branch and bound algorithms, to heuristics based on simulated annealing, neural nets, genetic algorithms, or ant colony optimization.

TSPLIB [1] provides TSP instances that are often used as benchmarks to evaluate the performance (in terms of both speed and quality of results) of the different heuristics. A good comparative survey and up-to-date picture of the state of the art in TSP heuristics can be found at the DIMACS Implementation Challenge [2] site.

Concorde [3] gathers many of these highly-optimized heuristics, including 2-Opt, in a single package. Several tricks are used to speed-up 2-Opt [4] [5]. Heavy pruning of legal moves, preprocessing to construct neighbor lists, *k-d* and other types of trees out of the cities, "*don't look*" bits, caching etc. are some of them. As a result of all this, the algorithm runs in subquadratic time (if not $O(NlogN)$, then at least no worse than $O(N^{1.2})$, where N is the number of cities) and is actually very fast for small TSP instances.

One often mentioned method for speeding up TSP is the use of parallelism. Geometric partitioning [7] uses rectangles to partition the set of cities into smaller subsets of neighboring cities, and sends the cities of each rectangle to a different processor. Another approach, tour-based partitioning [8] [9], partitions the current tour into segments, and sends each segment to a different processor. After all the sub-problems have been solved in parallel, they are combined to form a solution for the entire instance.

The partitioning of the cities performed by these algorithms, and the processor communication that is involved, incurs final tour quality loss and an extra cost, which is usually compensated for only in very large TSP instances, if at all. According to [5], the running time of [9] for a 11849-city instance run on 512 processors, is almost a factor 200 *greater* than the time for an optimized version of the 2-Opt algorithm running on a single processor, a factor that appears to be growing with instance size.

In contrast to the above parallel algorithms, our FPGA-based implementation is able to extract fine-grain parallelism even in small TSP instances and outperform the state-of-the-art software solution. Additionally, to the best of our knowledge, this is the first approach that evaluates and applies 2-Opt moves in parallel.

Almost all proposed algorithms and literature focus on software implementations of TSP heuristics. As far as we know, there has been very little work done on how these algorithms could be ported to hardware. A couple of hardware implementations of genetic algorithms for the TSP can be seen in the references [10] [11], but they are slower than the proposed solution since they yield significantly less parallelism.

3 Background on 2-Opt

The most famous and widely used tour modifications by local search algorithms are *2-Opt* and *3-Opt* moves. In this paper we will concentrate on the 2-Opt moves, even though our work can certainly be extended for 3-Opt moves. A 2-Opt move deletes two edges, thus breaking the tour into two parts, and then reconnects those paths in the other possible way (see Figure 1). This is equivalent to reversing the order of the cities between the two edges, thus a 2-Opt move can be seen as a segment reversal, and will be treated as such in this paper.

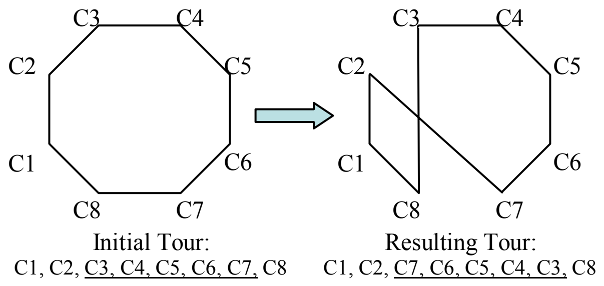

C3
C4
C2
C5
C1
C6
C8
C7
Initial Tour:
C1, C2, C3, C4, C5, C6, C7, C8
C3
C4
C2
C5
C1
C6
C8
C7
Resulting Tour:
C1, C2, C7, C6, C5, C4, C3, C8

Figure 1. Applying a 2-Opt move.

Notice that in Figure 1, reversing tour segment {C3..C7} is equivalent to reversing the remaining tour segment {C8, C1, C2}. Thus, in our search for length-reducing segment reversals, *we will only consider segments that consist of up to half the total number of cities.*

Each segment reversal consideration need only take into account *the four cities at the segment boundaries*, no matter how long the segment is (since all internal cities keep their relative order). For example, the difference in tour lengths that results from applying the move of Figure 1, can be calculated as:

$$\text{Delta(length)} = \text{dist}(C2,C7) + \text{dist}(C3,C8) - \text{dist}(C2,C3) - \text{dist}(C7,C8) \quad (1)$$

where *dist(A,B)* is the Euclidean distance between cities A and B, i.e.:

$$\text{dist}(A,B) = \text{sqrt}[(Ax - Bx)^2 + (Ay\text{-}By)^2] \quad (2)$$

If equation (1) turns out to be negative, this is a length-reducing move and should be applied to the current tour.

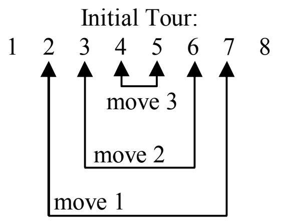
4 Symmetrical 2-Opt Moves

A local search algorithm that uses 2-Opt moves typically evaluates a significant number of moves before finding a move that reduces the length of the current tour. This is especially true as the algorithm approaches a local minimum solution, where hundreds or even thousands of moves might need to be evaluated before a length-reducing move can be found. These searches can be performed in parallel.

We define a set of segments (or equivalently their corresponding 2-Opt moves) as *symmetrical segments* (or *symmetrical 2-Opt moves*), if and only if:

- *For each segment SegmA of the set (except the smallest one), segment SegmB that consists of the same cities except the two cities at the boundaries of SegmA, is also part of the set.* For example, if SegmA = {C10..C100} is in the set, then SegmB = {C11..C99} will also be in the set. Taking this further, if we assume that SegmA is the largest segment in the set, then the set will consist of segments {C10..C100}, {C11..C99}, {C12..C98}, etc., down to whichever segment is the smallest.

Symmetrical 2-Opt moves are ideal candidates for parallel evaluation and application in hardware (for reasons that will become clear in the next paragraphs).



Moves applied		Resulting Tour:							
none		1	2	3	4	5	6	7	8
1		1	**7**	**6**	**5**	**4**	**3**	**2**	8
2		1	2	**6**	**5**	**4**	**3**	7	8
3	⇨	1	2	3	**5**	**4**	6	7	8
1,2		1	**7**	3	4	5	6	**2**	8
1,3		1	**7**	**6**	4	5	**3**	**2**	8
2,3		1	2	**6**	4	5	**3**	7	8
1,2,3		1	**7**	3	**5**	**4**	6	**2**	8

Figure 2. Example of three symmetrical 2-Opt moves.

Figure 2 shows an example where three symmetrical 2-Opt moves are evaluated in parallel; the first one considers the reversal of the tour segment {2..7}, the second considers segment {3..6}, and the third one considers segment {4, 5}. The Figure shows the starting tour, as well as the resulting tour after applying any of these moves. Underlined in the resulting tour are the cities that have changed positions, compared to the positions they had in the initial tour. For example, in the case where moves 1 and 2 are applied, segment {3..6} will be reversed twice, ending up at its initial position and orientation. Thus, in this case, only cities 2 and 7 will end up in different positions than their initial ones.

Looking at this Figure, the following observations can be made for a set of *symmetrical* 2-Opt moves:

1. After the application of *any number* of the symmetrical segment reversals under consideration, each city will either remain at its initial position, or swap positions with its "*symmetrical*" city (two cities are "*symmetrical*" if they are at the two ends of one of the symmetrical segments). For example, cities 2 and 7 of Figure 2 will either remain at their initial positions or swap positions (the same is true for cities 3 and 6, as well as cities 4 and 5).
2. Whether the cities at the two ends of a specific segment will stay put or will swap positions depends only on whether an even or odd number of segment reversals are applied on them. Therefore the number of the segment reversals can be counted by considering *only the segments that these cities are part of*. For example, in the case of Figure 2:
    - Cities 2 and 7 will swap positions iff segment {2..7} gets reversed.
    - Cities 3 and 6 will swap positions iff exactly one of the segments {2..7} and {3..6} gets reversed (not both).
    - Cities 4 and 5 will swap positions iff an odd number of the segments {2..7}, {3..6} and {4, 5} get reversed.

What these observations tell us is that, we are able to apply *any subset* of symmetrical 2-Opt moves *in parallel, just by figuring out which cities need to be swapped*. In what follows we will examine a hardware implementation that takes advantage of this *parallelism in applying* symmetrical 2-Opt moves. To the best of our knowledge, *this is the first attempt to exploit this type of fine-grain parallelism in the 2-opt algorithm.* Most attempts use a coarser-grain parallelism, by partitioning the cities in a geometric or tour-based fashion among different processors, and then combining their results.

5 Architecture

In order to take advantage of the aforementioned parallelism, the proposed architecture splits all legal 2-Opt moves into groups of symmetrical 2-Opt moves. The moves of each such group are then evaluated and those that actually turn out to be length-reducing are applied in parallel.

As an example, let us consider an initial tour that consists of 200 cities. Figure 3 shows the proposed architecture for this example. Each city is represented by a register that holds its two coordinates, and the cities are put one next to the other to form a 200-entry circular shift register. The order of the cities in this shift register represents the current tour. The architecture uses 50 Processing Elements (PEs) for the 200 cities of this example, and in general uses N/4 PEs for N cities (we will later describe an enhancement that relaxes this limitation), for reasons that will become clear as soon as we explain how it works. Arrows in the Figure show all possible data movements (datapaths).

Since we have 200 cities, and based on the reasoning of Section 3, we only need to consider segments of length at most 100. This is a total of 199 * 200/4 = 9950 such segments (half the combinations of picking 2 segment ends out of 200 cities). The proposed architecture will split these 9950 segments into 199 groups of 50 *symmetrical* segments each, and evaluate the segment reversals of each such group in parallel. The actual implemented algorithm executes the following tasks:

1. Firstly, it evaluates in parallel the set of 50 symmetrical segments {1..100}, {2..99}, {3..98}, etc., up to {50, 51}. These are all segments with even lengths of 2, 4, 6, etc., up to 100.
2. Secondly, it evaluates in parallel the set of 50 symmetrical segments {1..101}, {2..100}, {3..99}, etc. up to {50, 52}. These are all segments with odd lengths of 3, 5, 7, etc., up to 101.
3. Finally, the above two steps are re-applied so as to evaluate all remaining 197 groups of 50 symmetrical segments each.

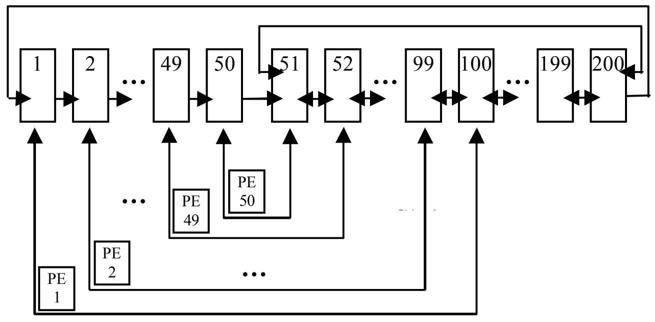


Figure 3. Architecture using 50 PEs for 200 cities.

For the first step, the 50 Processing Elements (PEs) shown in Figure 3 all run in parallel, each one evaluating one 2-Opt move. PE[1] evaluates whether the tour segment {1..100} should be reversed, PE[2] evaluates segment {2..99}, PE[3] evaluates segment {3..98}, etc., up to PE[50] which evaluates segment {50, 51}. In order for each PE to evaluate whether its 2-Opt move is length-reducing or not, as we discussed earlier, it needs to calculate equation (1) (see Section 3) for the cities at its segment boundaries. For example, PE[2] calculates equation (1) for cities 1, 2, 99 and 100. This is done in *O(1)* time (a few clock cycles) with the use of minimal hardware (a few multipliers and adders) per PE.

In this way, all PEs evaluate in parallel their 2-Opt moves, and when done, we will have 50 yes/no decisions for the 50 segment reversals under consideration. Next, we need to apply these decisions by swapping the necessary city pairs, as was explained in the previous Section. For example, cities 50 and 51 will swap positions iff PEs 1 to 50 have resulted in an odd number of "yes" decisions. Similarly, cities 49 and 52 will swap positions iff PEs 1 to 49 have resulted in an odd number of "yes" decisions, etc. The necessary XORing of the PE decisions can take as few as *log_X(number of PEs)* clock cycles, assuming we can XOR up to x PE decisions per cycle (though not *O(1)*, still a small number of cycles even for large TSP instances – x was equal to 5 in our FPGA implementations of Section 6). All necessary city swaps are done simultaneously in a single clock cycle, using the datapaths shown in the Figure.

Now, for the second step (segments with odd lengths 1, 3, 5, 7, etc., up to 101), in order to reuse the same wiring and PEs as before, we perform a left circular shift to segment {51..200} (see the corresponding datapaths in the Figure). In this way PE[1] will now be able to evaluate segment {1..101}, PE[2] will now evaluate segment {2..100}, PE[3] will evaluate segment {3..99}, etc., up to PE[50] which will evaluate segment {50..52}. After all these 50 new moves have been evaluated, we apply whichever turn out to be length-reducing by swapping the necessary cities as before (this time the swapping city pairs are different than before; city 50 with 52, city 49 with 53 etc.). Next, we perform a right circular shift on segment {52..200, 51} to restore city 51 to its original position. Notice that this scheme works since no matter how many of the segments under consideration are actually reversed, city 51 is in the centre of all these segments and will not change position.

For the third and final step, we perform a right circular shift to all 200 cities (see the corresponding datapaths in the Figure), and all the above tasks are again repeated from the beginning. In effect, the same wiring and PEs will now be used in a different position of the tour. This whole process is repeated until we can not find any length-reducing moves for 199 consecutive repetitions, since the algorithm is exhaustive and is *guaranteed* to search all possible tour segments in 199 repetitions (in contrast to randomized search algorithms that search 2-Opt moves at random and can not thus guarantee 2-Optimality of the final result).

Following is the pseudo-code for the control of the proposed architecture, which can be implemented with a simple Finite State Machine (FSM):

```
Repeat
    1.  PEs evaluate moves (segments with even lengths)
    2.  Apply length-reducing moves by swapping cities
    3.  Apply left circular shift to segment {51..200}
    4.  PEs evaluate moves (segments with odd lengths)
    5.  Apply length-reducing moves by swapping cities
    6.  Apply right circular shift to segment {52..200, 51}
    7.  Apply right circular shift to all cities
until done
```

Figure 4. Pseudo-code for the Architecture control.

Notice that the proposed architecture exhibits minimal and fixed wiring with trivial control. Furthermore, notice that, since all move evaluations and applications are done in parallel, each iteration of the above pseudo-code loop (all 7 steps) is executed in *O(1)* time (excluding the small number of cycles required for the XORing of the PE decisions), independent of the total number of cities (200 in this case).

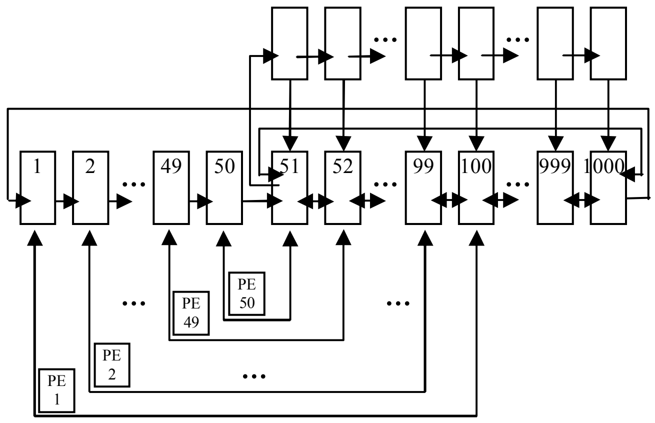


Figure 5. Architecture using 50 PEs for 1000 cities.

It is easy to see how this architecture scales to more cities, as long as there is enough silicon to accommodate the additional PEs and city registers. However, the same architecture, with a slight modification, can also work if we scale up the number of cities while keeping the number of PEs constant (thus using less than N/4 PEs for N cities), assuming PEs are expensive. In this case, we would still like to use the available PEs and their fixed wiring to evaluate all possible 2-Opt moves of our larger tour. Figure 5 shows an example with 50 PEs and 1000 cities. Notice the added registers and data paths in comparison to the architecture of Figure 3.

In order to allow the PEs to work on larger segments we will use the same technique as before, i.e. perform left circular shifts to segment {51..1000} for a number of cycles which depends on the actual length of the segments under evaluation. For example, by performing 50 shifts, PE[1] will evaluate segment {1..150}, PE[2] will evaluate segment {2..149}, etc., up to PE[50] which will evaluate segment {50..101}. These segments have lengths of 150, 148, etc., down to 52 cities respectively. After all these new 50 moves have been evaluated and applied (by swapping the necessary cities), we need to restore segment {51..100} from the end of the 1000-city wide shift register back to its original position. This is done by performing 50 right circular shifts to segment {101..1000, 51..100}. This time however, there is an extra step since segment {51..100} might need to be reversed (if cities 50 and 101 have been swapped). For this reason, we have added the extra set of registers (see Figure 5) that get loaded during the left circular shifts of segment {51..1000}, and will thus end-up holding the reverse of segment {51..100}. These registers are used as a final step to overwrite segment {51..100} with its reverse, if needed, in a single cycle.

Using the technique described we are able to use the same PEs and wiring to perform 50 move evaluations per iteration. A total of 10 iterations would be needed to be able to evaluate moves of all segment lengths from 2 up to 500 (we do not need to evaluate bigger segments for 1000 cities). Then, a right circular shift should be applied to all the cities before the algorithm can be repeated.

6 FPGA-Based Implementation

Several instances of the first version of the proposed architecture (the one in Figure 3), with varying number of cities, were implemented in Verilog and synthesized for the Xilinx xc2vp100-6 Virtex II Pro FPGA.

For these implementations, the PEs did not include hardware to compute the square roots of equation (2) – this is part of our ongoing and future work. For this reason, the Verilog simulations actually find the tour with the (locally) smallest sum of *squares* of distances between the cities. However, as we discuss in the following Sections, we have built an accurate software emulator of the architecture (that calculates all the required square roots) which allows us to accurately evaluate the architecture performance both in terms of running time and in terms of quality of resulting tours.

The main hardware units used, excluding pipeline registers, control and other miscellaneous logic, are the following:

- One 30-bit wide register per city to hold its X and Y coordinates (thus each coordinate can be up to 15 bits long, which was enough for all the TSPLIB instances that we use).
- One 16x16 multiplier, as well as one 16-bit and one 32-bit adder, per PE.

Having the PEs use the aforementioned hardware in a pipelined fashion, we were able to execute each iteration of the pseudo-code loop of Figure 4 in just 35 clock cycles (with each move evaluation taking 12 clock cycles).

Table 1 contains information about the area occupied by these implementations and their clock frequencies. The suffix of the TSPLIB instance name indicates the number of cities in that particular instance.

Table 1 Implementation Results for Virtex2Pro FPGA

Cities (TSPLIB instance)	*Area (slices)*	*Multipliers (MULT18X18s)*	*Speed (MHz)*
berlin52	3295 (7%)	13 (2%)	184
eil76	4823 (10%)	19 (4%)	178
rd100	6332 (14%)	25 (5%)	184
ch150	9390 (21%)	37 (8%)	165
ts225	14143 (32%)	56 (12%)	171
pr299	18749 (42%)	74 (16%)	165

7 Deterministic Nature of Architecture Algorithm

The proposed architecture evaluates 2-Opt moves in a strictly deterministic fashion, as opposed to a randomized local search algorithm that evaluates 2-Opt moves at random. A question that arises is how this loss of randomness affects the time for the algorithm to converge to a locally optimal tour, as well as the quality of this resulting tour.

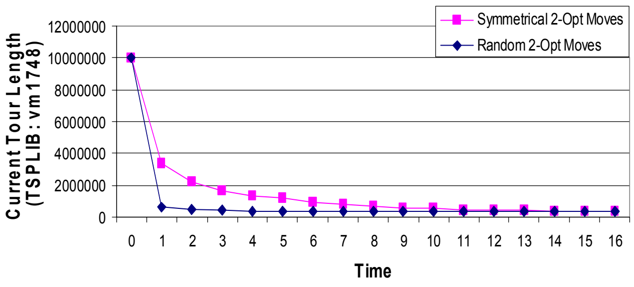
We implemented in software an *iteration-accurate* emulator of the proposed architecture (the version shown in Figure 3). The emulator executes the pseudo-code of Figure 4, searching the same groups of symmetrical 2-Opt moves in the exact same order as the architecture. We used this software emulator for all our performance measurements, due to the following reasons:

- The final tour, as well as all the intermediate tours at the pseudo-code loop iteration boundaries (i.e. at the beginning of each iteration), matched exactly with the ones from the Verilog simulations (after turning the square root calculations of the emulator off). This validates that the emulator *evaluates and applies the exact same 2-Opt moves* as our architecture.
- Since the software emulator executes the exact same number of pseudo-code loop iterations as the implemented hardware model, and each iteration takes *O(1)* time to execute in hardware, the emulator is able to *accurately predict the performance* of a hardware implementation of the architecture, as long as it knows how long *O(1)* actually is. The accuracy of the software emulator was verified against all the synthesizable hardware (Verilog) models of the previous Section.
- The software emulator runs orders of magnitude faster than a Verilog simulation.

In order to evaluate the effect of the deterministic nature of the architecture we also implemented in software a randomized version of the local search algorithm. This version evaluates 2-Opt moves at random, and converges after a sufficiently large number of sequential move evaluations (equal to the square of the number of cities) fail.

The two implementations were run against 70 TSPLIB instances (we used the Euclidean instances of "EUC_2D" type), *using the exact TSPLIB instances as our starting tours*, as opposed to first applying a greedy algorithm to them as is often done. The sizes of these instances ranged from 50 to around 4500 cities.

In these runs, our algorithm evaluating symmetrical 2-Opt moves converged, most of the times, slower than the algorithm that evaluated 2-Opt moves at random. Figure 6 provides a close look at the runtime behavior of the two algorithms for two specific TSPLIB instances (the results shown are typical at least for the set of TSPLIB instances that we used).



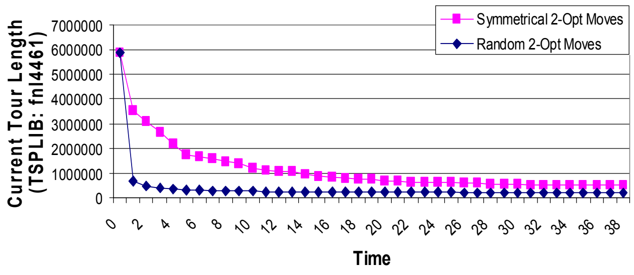


Figure 6. Runtime behavior of local search algorithm evaluating symmetrical or random 2-Opt moves.

However, the running time of a hardware implementation of our deterministic algorithm, is expected to be orders of magnitude lower than the running time of a hardware implementation of the randomized algorithm (see following Section for detailed performance results of the former). The reason is that while the former was designed with such an implementation in mind and exhibits great parallelism, the latter (even though we are not aware of such an implementation) would probably need to evaluate and apply the 2-Opt moves in a serial fashion.

As far as the final tour quality is concerned, the results of the two algorithms are pretty much the same (our deterministic algorithm produced an average of 0.75% smaller final tours than the randomized algorithm).

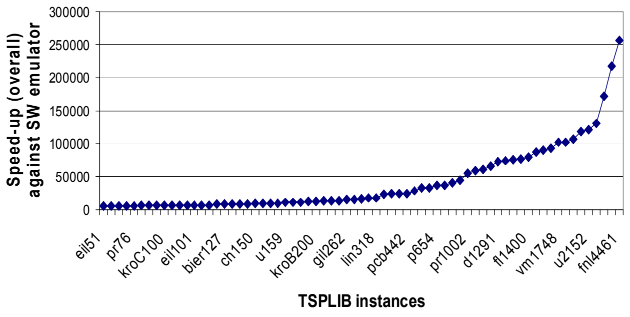
8 Performance Results

As we discussed in the previous Section, the architecture emulator is able to accurately predict the performance of a hardware implementation of the architecture, as long as it knows the duration of each iteration of the pseudo-code loop. For the performance results in this Section we have assumed that each iteration takes 35 clock cycles of a reported 150 MHz clock. Notice that after the hardware for the square root calculations is added to the PEs (in our future work), the number of cycles is expected to increase, which will have a negative impact to the performance numbers shown in this Section. However, the square root calculation can be performed by several estimation algorithms and thus the number of clock cycles needed is expected not to increase significantly.

We compared the performance of the hardware implementation of the architecture against two software implementations:

1. The architecture emulator.
2. A modified version of the 2-Opt local search algorithm implementation by Concorde. Concorde is widely considered as the state-of-the-art in TSP solving software, containing highly optimized implementations for the most important TSP algorithms and heuristics. We modified its 2-Opt algorithm implementation in the following way:
    - Concorde will by default first apply the greedy algorithm to the initial tour in order to obtain the starting tour for 2-Opt optimization. Since for the performance measurements of our architecture (and its software emulator) we used the exact TSPLIB instances as our starting tours, we had to modify Concorde so as not to apply the greedy algorithm to the initial tour.

Concorde runs very fast for the small TSPLIB instances that we consider. Thus, in order to more accurately measure its performance, we measured for each TSPLIB instance the average running time among 1000 runs.



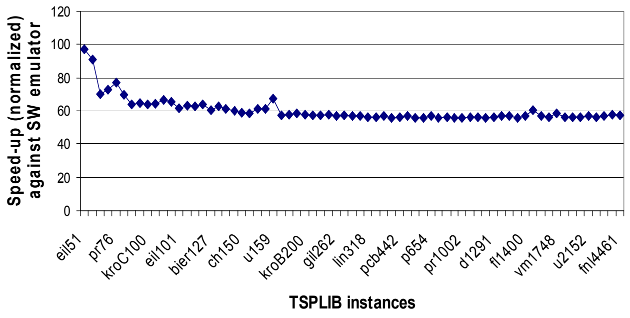


Figure 7. Overall and normalized speed-ups over software emulator (based on 35 150MHz ccs per iteration).

Both software implementations were run on an Intel Pentium 3 GHz machine running RedHat Linux. Figure 7 shows the speed-ups obtained by the hardware implementation against its software emulator, as well as the same speed-ups normalized to (i.e. divided by) the number of cities.

As expected the overall speed-up is proportional to the number of cities, since the latter reflects the number of PEs operating in parallel. The normalized speed-up of 60 seen in this Figure is attributed to the faster evaluation and application of 2-Opt moves in hardware over software.

Figure 8 compares the quality of the final tours obtained with our approach against the ones obtained by Concorde. Our architecture outperforms Concorde in final tour quality by an average of around 10%. This is expected since Concorde, in contrast to our architecture, gives up the guarantee of true 2-Optimality in favor of greatly reduced running time. The difference of 10% is exaggerated by the fact that we do not apply the greedy algorithm to our initial tours. However, it still shows the high quality of the tours produced by our architecture.

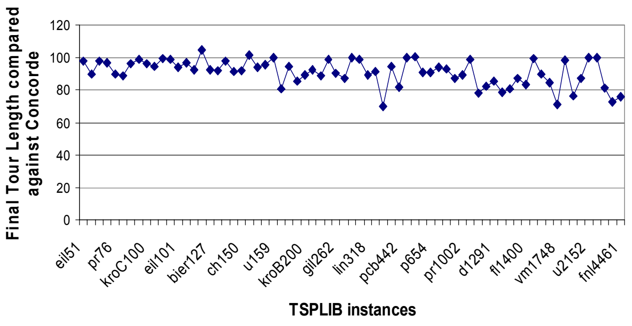


Figure 8. Final tour lengths achieved by our architecture as percentages of the ones achieved by Concorde.

Figure 9 shows the speed-ups that our architecture exhibits against the state-of-the-art Concorde. We can see that our architecture exhibits an average speed-up of around 6 for the TSPLIB instances that we tried. Notice the huge difference between these speed-ups and the ones of Figure 7, which attests to the high quality of Concorde's heuristics and optimizations over our simplistic software implementations. Additionally, Concorde's performance certainly scales better with the number of cities than our implementations.

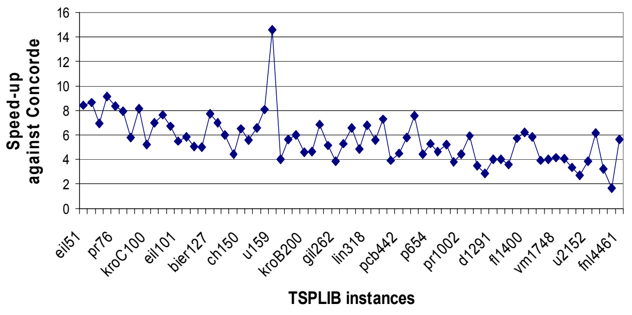


Figure 9. Speed-up over Concorde's 2-Opt implementation (based on 35 150MHz ccs per iteration).

Table 2 has a more detailed analysis of how the numbers for a small subset of the TSPLIB instances of Figure 9 were obtained.

Last but not least, we should note that greater speed-ups are achievable if, like Concorde, we give up the guarantee of true 2-Optimality. In our runs we observed that the algorithm using symmetrical 2-Opt moves converged to within 5% of the final locally optimal tour at usually 20% to 70% of its total running time. If we add a mechanism to stop running when small reductions in tour length are detected, we can gain additional speed-ups of 150% to 500%.

Table 2 Comparison between Concorde and our proposed architecture.

	Concorde	*Proposed Architecture*			
TSPLIB instance	Time (us)	Loop Iterat-ions	Clock cycles	Time (us)	Speed-up
eil51	460	234	8190	54.6	8.42
tsp225	1800	1129	39515	263.4	6.83
pr1002	5720	5576	195160	1301	4.40
rl1304	17380	18607	651245	4342	4.00
vm1748	30800	31747	1111145	7408	4.16
u2319	12330	8585	300475	2003	6.16
fnl4461	111350	84940	2972900	19819	5.62

9 Future Work

Regarding our ongoing and future work we are/will be working on the following tasks:

- Add hardware to the PEs for the square root calculations and adjust our performance numbers accordingly. In order not to add major hardware resources nor significantly increase the number of clock cycles needed for the PEs we will use a square root estimation technique such as the one in [12].
- Examine how our architecture scales for TSP instances larger than the ones considered in this paper, and explore the possibility of using external RAM.
- Explore the cost and performance of using our architecture to perform the Simulated Annealing algorithm using 2-Opt moves.
- Explore how we could handle 3-Opt moves.

10 Conclusion

In this paper we present an architecture tailored to reconfigurable hardware that evaluates and applies 2-Opt moves in a highly parallel manner, thus offering dramatic speed-ups to the 2-Opt local optimization algorithm. The proposed architecture is implemented on a state-of-the-art FPGA for TSP instances up to a few hundred cities, and produces better quality results, while it is on average around 600% faster than the state-of-the-art Concorde software package.

Acknowledgements

This work has been partially supported by the General Secretariat for Research and Technology of Greece through PENED 2003 programme, contract no. 03ED851, co-funded by the European Social Fund (75%) and National resources (25%).

References

[1] TSPLIB is a database of instances for the TSP and is currently available from http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[2] DIMACS Implementation Challenge on the STSP at http://www.research.att.com/~dsj/chtsp/

[3] Concorde is computer code for the TSP and is currently available for download from http://www.tsp.gatech.edu/concorde.html

[4] D. Johnson and L. McGeoch, ``*Experimental analysis of heuristics for the STSP*" chapter of "*The Traveling Salesman Problem and Its Variations*", Boston 2002, pp. 369-443. Draft of chapter currently available from http://www.research.att.com/~dsj/papers/stspchap.pdf

[5] D. Johnson and L. McGeoch, "*The Traveling Salesman Problem: A Case Study in Local Optimization*" chapter of "*Local Search in Combinatorial Optimization*", London 1997, pp. 215-310.

[6] "*Simulated Annealing Methods*" chapter 10.9 of "*Numerical Recipes in C*". Chapter available from http://www.nrbook.com/a/bookcpdf.php

[7] R.M. Karp, *"Probabilistic analysis of partitioning algorithms for the traveling-salesman in the plane"*, Math. Oper. Res. 2 (1977), pp. 209-224.

[8] J.R.A Allwright, D.B. Carpenter, *"A distributed implementation of simulated annealing for the traveling salesman problem"*, Parallel Computing 10 (1989), pp. 335-338.

[9] M. G.A Verhoeven, E.H.L. Aarts, P.C.J. Swinkels, *"A Parallel 2-opt algorithm for the Traveling Salesman Problem"*, Future Generation Computer Systems 11 (1995), pp. 175-182.

[10] I. Skliarova and A. Ferrari, "*FPGA-Based Implementation of Genetic Algorithm for the Traveling Salesman Problem and Its Industrial Application*", IEA/AIE 2002, LNAI 2358, pp. 77-87, 2002.

[11] P. Graham and B. Nelson, "*A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2*", Brigham Young University

[12] Xiaojun Wang, Brent Nelson, "*Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs*", FCCM 2003.

[13] Ioannis Mavroidis, Ioannis Papaefstathiou, Dionisios Pnevmatikatos, "*Hardware Implementation of 2-Opt Local Search Algorithm for the Traveling Salesman Problem*", Technical University of Crete, Greece, International Workshop on Rapid System Prototyping 2007.