Application Specific Computers for Combinatorial Optimisation

Abramson, David; Logothetis, Paul; Postula, Adam; Randall, Marcus

Published in:
Proceedings of Computer Architecture

Application Specific Computers for Combinatorial Optimisation

David Abramson †
Paul Logothetis †
Adam Postula §
Marcus Randall †

† School of Computer and Information Technology,
Griffith University, Brisbane.
davida@cit.gu.edu.au

§ Department of Electrical and Computer Engineering,
University of Queensland, Brisbane.

Abstract: Solving large combinatorial optimisation problems is often time consuming, and thus there is interest in accelerating current algorithms by building application specific computers. This paper focuses on accelerating general local search meta-heuristics, such as simulated annealing and tabu search, and presents an architecture for this class of algorithms. As a design case study we describe a specific machine which implements a simulated annealing based algorithm for solving the Travelling Salesman Problem (TSP). This processor which is built with XILINX field programmable logic and APTIX interconnection matrices achieves a speedup of about 37 times over an IBM RS6000 workstation. The paper highlights the use of reconfigurable memory as a key for obtaining high performance.

1. INTRODUCTION

Application specific computers (ASCs) differ from general purpose ones in that they usually only solve a small range of problems, but often at much higher rates and lower cost. Their internal structure is tailored for the particular problem, and thus can achieve much higher efficiency and hardware utilisation than a processor which must handle a wide range of tasks. Over the years, many ASCs have been built to address a wide range of disciplines, ranging from image processing to encryption. In recent years, the advent of programmable logic devices has made it much easier to construct complete systems, and thus there has been much more interest in such machines [8, 11]. A recent conference series has been devoted to the field of custom computing machines [21, 22, 23, 24].

Combinatorial optimisation problems (COPs) can be solved by a number of algorithms, ranging from exact methods, like branch-and-bound, through to heuristic approximation schemes. Whilst exact methods are attractive because they return a result which can be proved to be optimal, they often fail on large real world problems. Accordingly, heuristic methods are often used to find good solutions which can be applied in practice. Good performance is achieved for a number of heuristics which are tailored

for particular problems, but a range of meta-heuristics are often used for solving general problems. Simulated Annealing (SA) [36, 25, 14, 16, 26], Genetic Algorithms (GA) [19] and Tabu Search (TS) [16, 18, 29] are examples of modern meta-heuristics [30]. These algorithms can be quite slow on large problems, thus there is interest in supporting their execution with high performance computer systems, such as vector and parallel supercomputers [1, 5, 6, 7, 15, 20, 27, 30, 1, 2, 12], and application specific machines [15, 3, 32, 33].

In this paper we describe an architecture for an ASC which will can be used to implement a range of meta-heuristics, in particular simulated annealing and tabu search, on a range of COPs. This type of machine is targeted at rapid prototyping and evaluation of mathematical models. In this environment a user might prototype a model and wish to iteratively refine the model across different data sets. Accordingly, we have chosen a generic representation for COPs, and a set of neighbourhood operators which work for a range of local search heuristics. The generic representation, which is based on list data structures, allows a user to specify a problem in terms of a cost function and a set of constraints, in a similar manner to other vector based optimisation systems like GAMS [10] and AMPL.

To date, we have built a demonstration ASC which can solve COPs which have a particular structure. We demonstrate its ability to implement simulated annealing on the Travelling Salesman Problem (TSP). We then describe the changes which are necessary to expand the range of COPs which can be solved.

2. COMBINATORIAL OPTIMISATION PROBLEMS

2.1 Representing and solving COPs

There are many ways of representing general COPs, however, it is usually possible to express a problem in terms of minimising (or maximising) some function ($f(x)$) subject to a set of constraints ($g(x) > 0$). In most COPs, the solution is often encoded with vectors of integers or binary variables.

When both functions, $f(x)$ and $g(x)$ are linear, then linear programming methods can be used to solve the problem. A standard method of doing this is to repeatedly solve continuous relaxations of the discrete problem using the simplex method within a branch and bound search strategy [32]. This method has been applied to a very wide range of problems over the years, but, in general, only works for linear problems. When either or both of the functions are non-linear, or dis-continuous, as in the case of many COPs, then the problems are much harder to solve.

Accordingly, a range of heuristics have been devised for solving particular COPs. These heuristics move through the potentially enormous search space using local information to guide the choice of move. In this paper we consider simulated annealing and tabu search, which make use of local information plus randomisation to move. One of

the advantages of these meta-heuristics is that they can be applied to a wide range of problems without building in special knowledge about the problem structure. However, even though the search algorithm is general, in most cases, it is still necessary to write a specific program for each different problem, because the data structures vary widely between problems. In the next section we introduce a list based representation which can be used to model a range of problems. Using this general list structure we then develop a set of local search heuristics which can be used on a wide range of problems.

2.2 A List Representation

Many combinatorial optimisation problems can be expressed as some form of assignment of elements to a number of different groups. A very natural representation of the above is to use a dynamic list data structure which records the elements mapped to each group. A local search process then consists of a number of transitions, in which elements are moved between groups. Because the list structures are dynamic, they grow and shrink as the solution is modified. Such a representation can eliminate some of the complex encoding required by other systems such as 0-1 ILP (Integer Linear Program), which require a number of constraints for enforcing the encoding over and above the normal problem constraints.

The exact data structures required vary depending on the problem, however, in general the items being assigned are stored in data cells which are linked together. For example, if items are being assigned to a *position* in solution space, then an ordered list of items can be maintained, in which the ordinal of the cell represents its *position*. Further, if more than one item is assigned to a given position, then these can be linked into a list, which is then linked into the position list. Accordingly, multiple nested lists can be used to represent the solutions to a wide range of problems.

The simplest way to illustrate the list coding technique is to consider an example. In the Generalised Assignment Problem (GAP) [28], a set of *jobs* must be assigned to a set of *agents*. Jobs are assigned subject to a number of capacity constraints, such that the cost is minimised. Using a list notation, a solution for a 5 agent, 15 job problem is shown in Figure 1.

In this example, job 4 is assigned to agent 3 and job 13 is assigned to agent 5 etc. Agents are linked together so that the solution can be represented by a single list pointer to agent 1. The objective function is to minimise the total cost of assigning the jobs to the agents (or more generally elements to group) subject to constraints which specify that the agent capacities must not be exceeded, which is described by equations (1), (2) and (3). Equation (1) measures the cost of assigning jobs to agents. Equation (2) specifies that the agents have fixed capacities, and equation (3) specifies that a job can be mapped to only one agent.

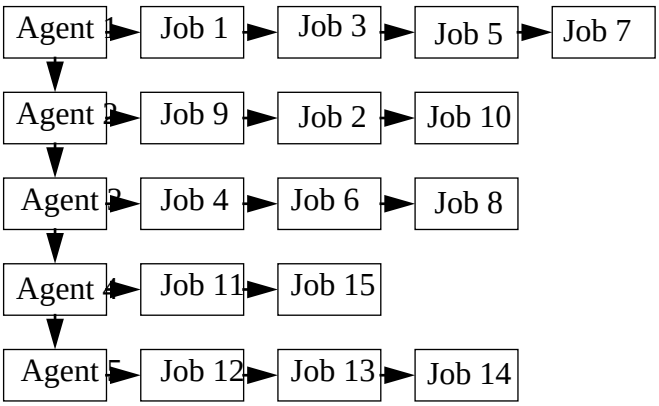

Fig 1. Sample list representation

$$\textit{minimise} \qquad \sum_{i=1}^{M} \sum_{j=1}^{|x_i|} C(x_{ij}, i) \tag{1}$$

$$\sum_{j=1}^{|x_i|} a(x_{ij}, i) \leq b(i) \qquad 1 \leq i \leq M \tag{2}$$

$$x_{ij} \neq x_{km} \qquad \begin{matrix} (1 \leq i, k \leq M) \\ i \neq k \end{matrix} \qquad \begin{matrix} (1 \leq j \leq |x_i|) \\ (1 \leq m \leq |x_k|) \end{matrix} \tag{3}$$

Where: x_{ij} is the j[th] job performed by agent i
$C(\varphi,\alpha)$ is the function that returns the cost of assigning job φ to agent α
M is the number of agents
$|x_i|$ is the size of the sub-list at position i
$a(\varphi,\alpha)$ is the amount of resource required for agent α to process job φ
$b(\alpha)$ is the capacity of agent α

The important features of this representation are that it is possible to represent a very wide range of common combinatorial optimisation problems, and as discussed in 2.4, a simple set of operators can be used to explore the neighbourhood of a given solution. This latter attribute makes the implementation of the algorithm feasible in hardware, and thus makes it possible to develop an application specific computer for solving COPs.

2.3 Local Search over Lists

Most local search algorithms can be summarised by a simple template which involves iterative updating of a feasible solution. The differences between the algorithms tend to

be in the way they define a neighbourhood of potential solutions, and the rules for updating the current solution. The following skeleton describes the basic operations of a local search algorithm:

```
X = Generate Initial Feasible Solution ()
Repeat
        Move = Select a Move from Neighbourhood (X)(1)
        X' = Apply Move(X,Move)(2)
        ΔC = Compute Change in Cost (X, X', Move)(3)
        If accept then X = X'(4)
Until
        Stopping Condition
```

The key calculations which are performed repeatedly by this algorithm are the generation of a move in the neighbourhood of the current one (1); application of the move to compute a new position, X' (2); the computation of the difference in cost between the new solution and the previous one (3), and whether to accept the change (4).

In simulated annealing the neighbourhood is a random perturbation of the current state [14, 16]. For example, it may be computed by choosing a random element of the solution, and changing it to another value. Step (4) is computed by evaluating Boltzmann's equation and comparing the probability of acceptance to a randomly chosen value between 0 and 1. Step (3) is typically problem dependent, as discussed in the previous section.

In Tabu search, steps (1), (2) and (3) are repeated for the entire neighbourhood of the current solution. In general, the acceptance rule is to accept the *best* of all of the neighbours of the current solution. However, a special list, called a tabu list, is maintained which holds some of the recent moves of the algorithm. If a potential move is in the tabu list then it is not selected, which prevents the algorithm from cycling through previous solution states, and also introduces some randomness into the search.

2.4 List Neighbourhood Operators

One of the most attractive features of the list representation discussed in section 2.2 is that relatively few transition operators are required to generate the neighbourhood. Further, the transition operators that can be applied for a particular problem can be chosen based on the types of the constraints and the cost function.

To date, our research has identified four main transition operators which are sufficient to solve a wide range of problems. The operators are:

Move	An item is moved from one list to the end of another list.
Swap	The position of two items, from the same or different lists, are swapped.

Inversion	The sequence between two items on the same list is reversed.
Reposition	The position of an item in a list is changed.

Importantly, the range of operators which can be applied to a particular problem can be determined based on the constraints which are specified. For example, if the constraint of the type $x_i \neq x_j$ is specified, then the *swap* or the *inversion* operators can be applied to reorganise the solution without violating these constraint.

It is possible that more than one operator may be applied to a given problem. For example, the *move* operator takes an item off one list and relocates it to the end of another. If the solution quality depends on order as well as position, then it might also be appropriate to apply the *reposition* operator, or the *swap* operator as well as the *move*. Importantly, each of these can be implemented using simple list management operations.

3. AN ARCHITECTURE FOR LOCAL SEARCH

In this section we describe an architectural template which is capable of applying the generic local search heuristic outlined in section 2.3 to a range of problems represented by lists. Figure 2 shows a basic machine structure, incorporating:

1. a unit for storing the current solution,
2. a unit for storing the new solution,
3. an update unit,
4. a change in cost generator,
5. a neighbourhood generator and
6. a unit for applying a move

The current solution is stored in high speed memory (1) and it is modified by the apply move unit (6). The neighbourhood generator (5) is responsible for deciding what type of local transition to perform and for generating the move. The output is a *move type descriptor*, which incorporates the kind of move and a set of index values. The change in cost unit (4) computes a change in cost based on the current solution and the new solution (2). If an incremental computation is possible it may also use the description of the move rather than recomputing the cost of the new solution ab initio. The update unit decides how to update the current solution. In Simulated annealing, it evaluates Boltzmann's equation decides whether to update the current solution. In Tabu search, it may store the best move which is not tabu. The architecture in Figure 2 implies that the new solution is copied to the old one, however, it is also possible to apply the move directly to the old solution. In the example considered later in the paper this optimisation means that a fast update of the solution can be applied.

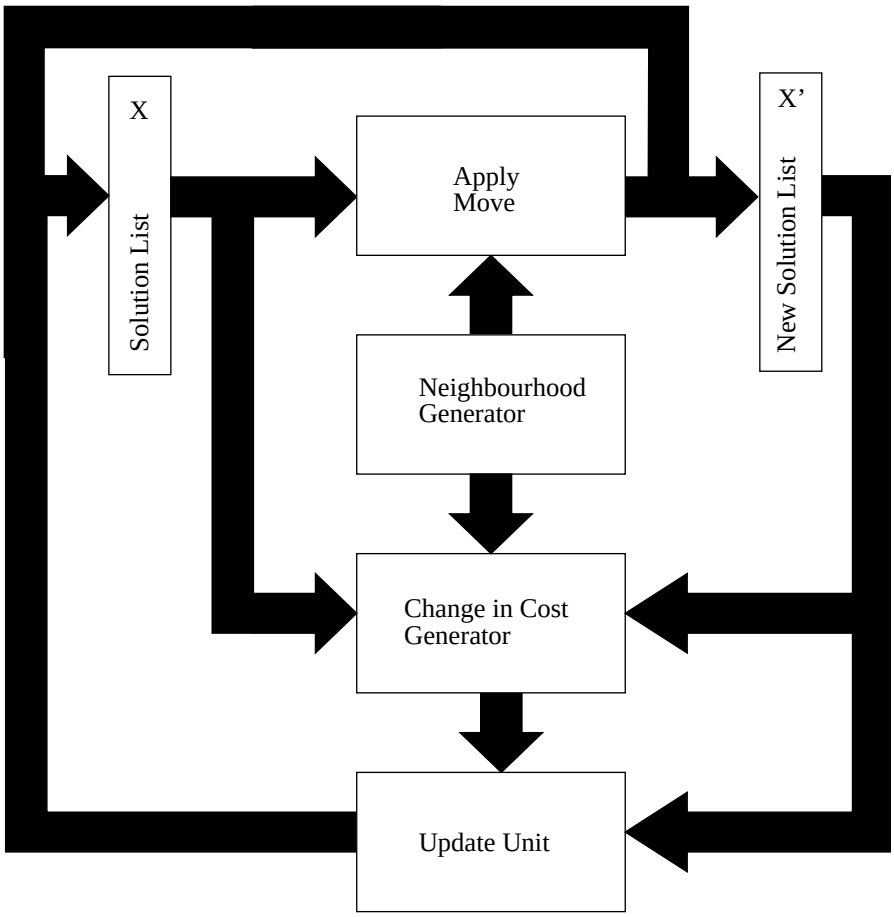

Fig 2. An Architecture for local search

An important feature of the machine in Figure 2 is that every unit except the change-in-cost generator can be generic across all problem descriptions. The change-in-cost generator must be able to compute the cost change which occurs as a result of a move. In one form it might re-evaluate the cost function for the new solution and subtract this from the current cost. However, for many problems it is possible to write an incremental cost formula which only uses the existing solution and the move descriptor to compute a change in cost. Because the cost function varies widely across problem type, it must be altered for each problem. In the next section we illustrate the type of function which can be written for the travelling salesman problem, and show how this can be efficiently supported by a set of nested memory devices.

4. A MACHINE FOR THE TRAVELLING SALESMAN

4.1 Mapping the TSP

In this section we present the design of a machine which can solve the Travelling Salesman Problem (TSP) using a simulated annealing search heuristic [9, 25]. TSP can be defined using the list representation of section 2.2 as follows:

minimise $$\sum_{i=2}^{N} d(x_i, x_{i-1}) + d(x_N, x_1)$$

subject to:

$$x_i \neq x_j \qquad (1 \leq i, j \leq N) \qquad i \neq j$$

Where:

x_i is the i^{th} city on the tour

$d(c_1,c_2)$ is the distance between city c_1 and city c_2

An important feature of this representation of the TSP is that it is possible to move through the search space either by swapping entries in the list, or by reversing a sub list. In this paper we will use the swap operator because it simplifies the problem representation and improves the parallelism in the cost computation, although, overall, it requires more hardware to compute the change in cost.

It is possible to compute an incremental change is cost when city i is swapped with city j, using the following formula:

$$\Delta C = (d(x_i, x_{i-1}) + d(x_i, x_{i+1}) + d(x_j, x_{j-1}) + d(x_j, x_{j+1})) \quad (4.1)$$

$$-(d(x_j, x_{i-1}) + d(x_j, x_{i+1}) + d(x_i, x_{j-1}) + d(x_i, x_{j+1})) \quad (4.2)$$

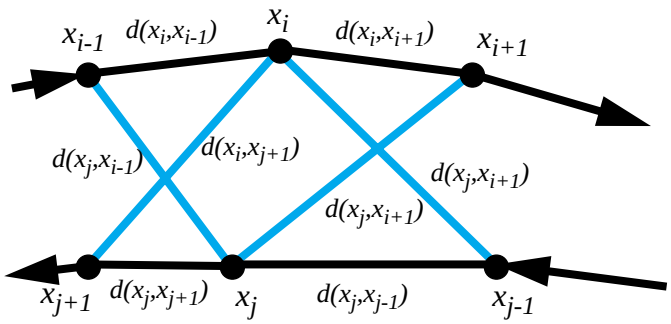

Fig 3. Effect of swapping two cities

Figure 3 shows the effect of swapping two cities, and explains the terms in equations 4.1 and 4.2. Since x_i and x_j are swapped, it is necessary to compute the distance to the cities which are adjacent, namely x_{i-1}, x_{i+1}, x_{j-1} and x_{j+1}. After the swap, city x_i is adjacent to city x_{j-1} and x_{j+1}, and city x_j is adjacent to cities x_{i-1} and x_{i+1}.

Whilst it is possible to evaluate this expression sequentially, it is faster to evaluate in parallel using an adder tree, as shown in Figure 4. In this diagram each of the 4 distance terms is added using a set of integer adders, and the old cost is subtracted from the new cost in one final step. Clearly, this arrangement can add the 8 numbers using 3 adder delays rather than 8. The scheme could be made even faster if necessary by using carry-save adders.

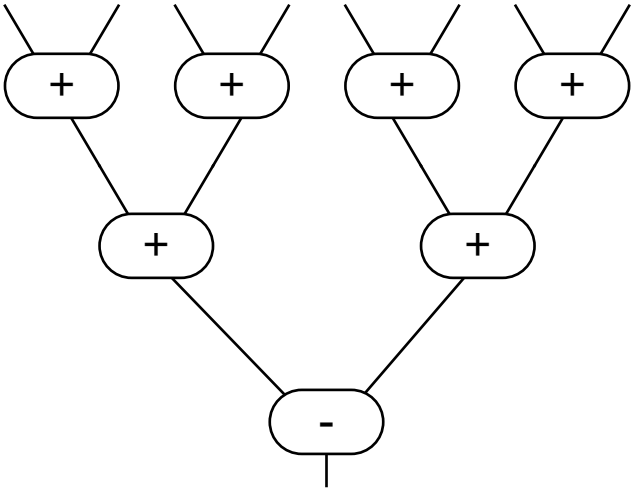

Fig 4. Adder tree for evaluating change in cost for TSP

Figure 5 shows the design of a machine which solves the TSP. As in Figure 2 it contains memory for storing a solution (A), a unit for generating a move (B), a unit for computing the change in cost (C) and an update unit which applies the Boltzmann equation (D).

Unit (B) is responsible for generating two city indexes, and for computing the neighbourhood of the cells. It does this by generating two different random numbers between 0 and N-1, and then computing the increment and decrement of each number modulo N. These 6 numbers are used as direct indexes for the tour memory (X), which is held in unit (A). If the change in cost is computed sequentially then only one copy of X is required. However, in order to use the adder tree shown in Figure 4 to add the distance measures in parallel, it must be possible to read the cities at each of the 6 positions concurrently. Accordingly, Unit (A) provides 6 identical copies of the X memory. The distances between adjacent cities is computed in two different ways. The terms in equation

4.1 pertain to part of the current tour, and those in equation 4.2 relate to the part in the new tour. It is not strictly necessary to recompute the terms of equation 4.1 because these can be stored from the last update of that section of the tour. In Unit (C) the parts of the old tour are stored in 4 identical copies of an inter-city distance memory, **d**. The four copies of **d** are indexed by i-1, i, j-1 and j to provide the distances between cities at position i-1 & i, i & i+1, j-1 & j and j & j+1. The inter-city distances in the new tour are computed by 4 lookup tables, D. Each D memory is addressed by the concatenation of two city identifiers, and presents the distance between those two cities. This approach works well for small problems because it avoids the need to compute the euclidian distance between cities. For larger problems we plan to explore alternative distance computation hardware. Each of the 8 distances is then presented to the adder tree and a change in cost is computed.

The change in cost is presented to the update unit (D), which evaluates Boltzmann's equation to decide whether the swap should be accepted. If ΔC is negative, then it is accepted without any further computation. However, if ΔC is positive, the expression $\exp(-\Delta C/T)$ is evaluated. To avoid having to calculate an exponential function, a lookup table containing the values of $\exp(-\Delta C/T)$ for various different values of ΔC is used. The table is reloaded each time the temperature is changed, which only occurs at the end of a Markov chain after several thousand swaps. This scheme was used in [3] and was quite successful because the table values for the next Markov chain can be computed by the host workstation in parallel with the execution of the current Markov chain.

If a swap is accepted, then the various memories must be updated accordingly. Cities at locations *i* and *j* must be swapped in the X memories, which takes two separate write operations. Since all copies of X are identical, it is possible to write to them all concurrently. Likewise, the **d** memories must be updated with the new inter-city distances, and 4 writes are required. Since the D memories store the distances between all possible cities, they need not be updated.

4.2 Implementation

The TSP machine is currently implemented on an Aptix AP4 reconfigurable logic board, containing a number of Xilinx XC4010 FPGAs and Aptix interconnection chips. The design is partitioned across 2 x XC4010s and a number of memory devices. The adder tree and update unit currently occupy one FPGA, and the remainder of the logic, including a finite state machine for control, is implemented in the other. The main requirement for splitting the design across FPGAs was due to pin limitations on the XC4010s rather than insufficient logic on the devices.

One of the more interesting features of the design is the split between high speed memory and logic. Whilst all of the logic can be accommodated on two FPGAs, a number of memory devices are required to store the multiple copies of the solution list and the distance matrix. Since the Xilinx FPGAs are not suitable for implementing large memories, we built a number of memory modules which can be inserted into an FPGA

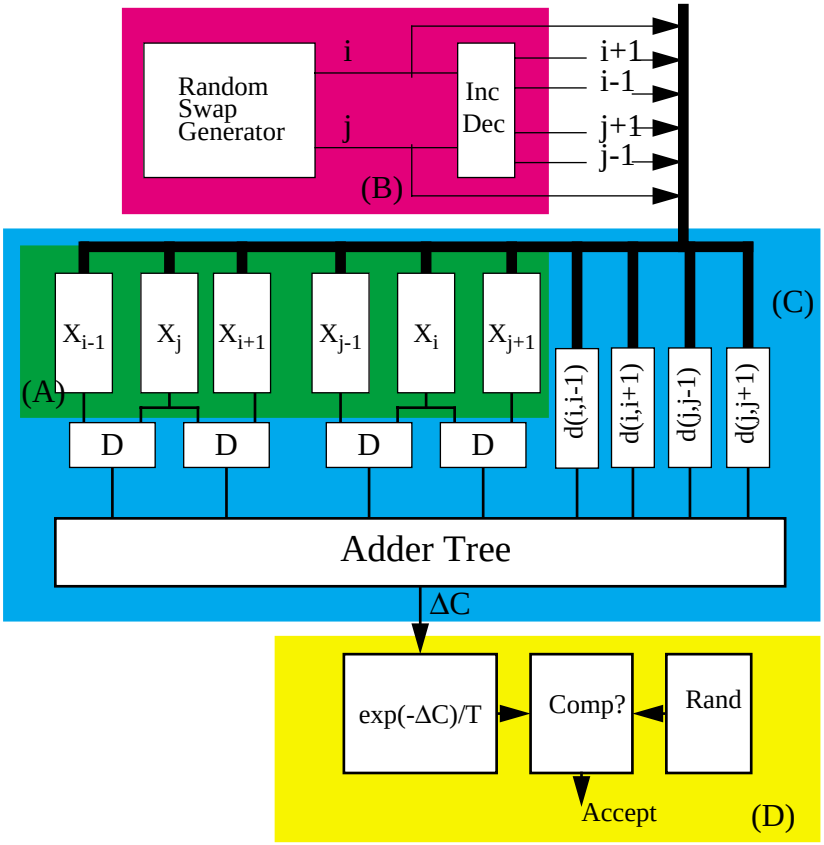

Fig 5. Architecture for solving the Travelling Salesman Problem

socket. Each module contains 5 x 128k x 8 bit, 20 nano-second, static memories and utilises about 150 pins on the FPGA pin grid array. The modules are then plugged into the AP4 board in place of FPGAs, and are interconnected using the Aptix switches. The design has highlighted the need for reconfigurable memories as well as logic in order to achieve high performance, which has been discussed elsewhere in the literature [11]. Due to limitations on the size of the D memories, the current system can only solve TSPs up to 256 cities. However, we envisage larger systems if different distance matrix implementation techniques are used.

The design is specified in a mix of schematic circuits and VHDL, which is then synthesised using the Autologic 2 tools from Mentor Graphics. Partitioning is performed by the Xilinx Foundary tool set, however, we found it necessary to provide information based on our knowledge of the problem to assist the partitioner.

Table 1 summarises the expected performance of the implementation based on detailed simulation results. This shows that the architecture should achieve a speedup of about 37 times over an RS6000 Model 590 workstation. We expect to improve the per-

formance significantly by tuning the design of the finite state machine which controls the system.

Table 1: Hardware and Software times

Implementation	Average Time per iteration
Software - IBM RS6000	13 μsecs
Hardware	0.35 μsecs
Speedup over RS6000	37 times

Table 2 shows the performance of the software implementation of the algorithm in a range of problems taken from TSPLIB [31]. It can be seen that the SA algorithm achieves optimal results on most problems. In the case of problem 3, the algorithm achieves a result within 0.09% of optimal but only after 9 hours of run time.

Table 2: TSP Problems and performance of algorithm

Problem	Number of Cities	Optimal Cost	Average time for 20 runs (s/w) (Second)	Average Gap (%)	Best Gap (%)
1	14	26	< 1	0	0
2	29	9063	45	0	0
3	48	33503	33609	0.32	0.09
4	52	7526	37	0	0
5	70	652	368	0	0
6	96	468	176	0	0

5. CONCLUSIONS AND FURTHER WORK

In this paper we have described a generic architecture for solving a range of combinatorial optimisation problems, and have illustrated it with a specific solution to the travelling salesman problem. Currently, the general architecture acts more as a template for designing a number of specific machines, rather than as an architecture which can be used to solve a range of problems without modification. Given the experience of designing the TSP machine, we are confident that it is possible to model a range of problems

in a generic form and then build one machine which can solve any of them. Specifically, a generic list structure memory will need to be developed, and a move generation unit which builds a *move descriptor*, as discussed in section 3. Also, the change in cost computation must be tailored for each problem, and thus a compiler will need to be developed for translating change in cost expressions into hardware. We plan to continue research in this area.

Further, we have only discussed the specific details of simulated annealing in the TSP machine. We are planning to implement a version of the machine which uses Tabu search, which will require the implementation of a Tabu memory as discussed in section 2.3. We plan to experiment with some different associative memory designs for tabu memories.

The implementation of the TSP machine highlighted a number of features of reconfigurable machines, most importantly, the need for reconfigurable memory devices. Accordingly, we built a memory unit which 'appeared' to our reconfigurable hardware as a FPGA, and thus we were able to take advantage of the Aptix switch hardware to connect the memories to the logic devices in a flexible way.

ACKNOWLEDGEMENTS

This project is funded by the Australian Research Council Large Grants Scheme. Thanks go to Mohan Krishnamoorthy from the CSIRO, and Yazid Sharaiha fro Imperial College for their helpful discussions.

REFERENCES

1. Aarts E.H.L, de Bont F.M.J., Habers J.H.A. and van Laarhoven P.J.M, (1986) "A Parallel Statistical Cooling Algorithm**",** *Proceedings STACS 86, Springer Lecture Notes in Computer Science*, **210**, 87-97, 1986.
2. Aarts E.H.L., de Bont F.M.J., Habers J.H.A. and van Laarhoven P.J.M., (1986) "Parallel Implementations of the Statistical Cooling Algorithm", *Integration*, **4**, 209-238, 1986.
3. Abramson D.A., (1992) "A Very High Speed Architecture to Support Simulated Annealing", *IEEE Computer*, May 1992.
4. Altman E, Marsland T and Breitkreutz, (1988) "Accounting for Parallel Tree Search Overheads", International Conference on Parallel Processing, 1988, Vol 3, pp 198-201
5. Arthur, J. L., Frendewey J. O. and Schumichrast, R. T. (1987) "Experience with Vectorizing a Linear Programming Code" (manuscript), Oregon State University, Corvallis, OR, 1987.
6. Arvindam S, Kumar V and Rao, N, (1990) "Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD", The Third Symposium on the Frontiers of Massively Parallel Computation, October 1990, University of

Maryland, pp 166-169.

7. Beasley, J. E. (1987) "Linear Programming on CRAY Supercomputers" (manuscript), Derpartment of Management Science, Imperial College London, England, Septemper 1987.

8. Bertin, P, Roncin and Vuillemin, (1990) "Programmable Active Memories: A Performance Assessment", Technical Report, DEC Paris Laboratories, 1990.

9. Bonomi E. and Lutton J.L., (1984) "The N-City Travelling Salesman Problem: Statistical Mechanics and The Metropolis Algorithm", *SIAM Review*, **26**, 551-568, 1984.

10. Brooke, A. Kendrick, D. and Meeraus, A. (1988) "GAMS: A user guide", The Scientific Press, CA, 1988.

11. Buell, D., Arnold, J and Kleinfelder, N. (1996) "Splash 2: FPGAs in a Custom Computing Machine", IEEE Press, 1996, ISBN 0-8186-7413-X.

12. Casotta A., Romeo F. and Sangiovanni-Vincentelli A.L., (1986) "A Parallel Simulated Annealing Algorithm for the Replacement of Macro-Cells", *Proceedings IEEE International Conference on Computer Aided Design*, Santa-Clara, 30-33, November 1986.

13. Cheng K and Wang Q, (1990) "An Asynchronous Multiprocessor Design for Branch-and-Bound Algorithms", The Third Symposium on the Frontiers of Massively Parallel Computation, October 1990, University of Maryland, pp 65-68.10

14. Collins N.E., Eglese R.W. and Golden B.L., "Simulated Annealing: An Annotated Bibliography", *American Journal of Mathematical and Management Sciences*, **8**(3-4), 209-307, 1988.

15. Edwards J. Tomlin, J. (1992) "Parallel Cholesky Factorization", Proceedings of 5th Australian Supercompting Conference, pp 105 - 114, December 1992.

16. Eglese R.W., (1990) "Simulated Annealing: A Tool for Operational Research", *European Journal of Operational Research*, **46**, 271-281, 1990.

17. Glover, F. (1989) "Tabu Search - Part 1", ORSA journal on Computing 1 (3), 190-206

18. Glover, F. (1990) "Tabu Search - Part 2", ORSA journal on Computing 2(1), 4-32

19. Goldberg, D. E. (1989) "Genetic Algorithms: In Search, Optimization and Machine Learning". 1989, Addison-Wesly Publishing Co.

20. Greening, D. (1989) "A Taxonomy of Parallel Simulated Annealing Techniques", UCLA Computer Science Department technical report number CSD-890050, August 21, 1989, Los Angeles, California.

21. IEEE Computer Society, (1993) "Proceedings of IEEE Workshop in FPGAs for Custom Computing Machines", April, 1993, Napa, California.

22. IEEE Computer Society, (1994) "Proceedings of IEEE Workshop in FPGAs

for Custom Computing Machines", April, 1994, Napa, California.

23. IEEE Computer Society, (1995) "Proceedings of IEEE Workshop in FPGAs for Custom Computing Machines", April, 1995, Napa, California

24. IEEE Computer Society, (1996) "Proceedings of IEEE Workshop in FPGAs for Custom Computing Machines", April, 1996, Napa, California

25. Kirkpatrick S., Gelatt Jr. C.D. and Vecchi M.P., (1993) "Optimization by Simulated Annealing", *Science*, **220** 671-680, 1993.

26. Koulamas C., Antony S. R. and Jaen R., (1994) "A Survey of Simulated Annealing Applications to Operations Research Problems", *Omega*, **22**(1), 41-56, 1994.

27. Ortega, J. (1988) "Introduction to Parallel and Vector Solutions of Linear Systems", Frontiers of Computer Science Series, Plenum Press, 1988.

28. Osman, I. H., (1993) "Heuristics for the Generalized Assignment Problem", *Working Paper*, Institute of Mathematics and Statistics, University of Kent, Canterbury, 1993.

29. Osman, I. H., (1993) "Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem", *Annals of Operations Research*, **41**, 421-451, 1993.

30. Osman I.H. and Laporte G., (1996) "Metaheuristics: A bibliography", *Annals of Operations Research*, **63**, 513-623, 1996.

31. Reinelt, G. (1991) "TSPLIB - a travelling salesman problem library", ORSA Journal on Computing, vol. 3, no. 4 (1991) 376-384

32. Salami, M and Cain, G., (1996) "Genetic Algorithm Processor on Reprogrammable Architectures", Proceedings of the Fifth Annual Conference on Evolutionary Programming 1996 (EP96), MIT, Press, San Diego.

33. Salami, M and Cain, G., (1995) "A Multiple Genetic Algorithm Processor for a PID Controller System", Proceedings of the International Conference on Genetic Algorithms 95, (MENDEL95), University of Brno, Czech Republic, Sept 1995, pp 67-71.

34. Taha, H., (1992) "Operations Research: An introduction", Macmillan Publishing Company, New York, 5th Ed, 822 pages, 1992.

35. Thomborson, C. (1993) "Does your workstation computation belong on a supercomputer?", Comms. of A.C.M., November 1993, Vol 36, No. 11, pp 41-49.

36. van Laarhoven P.J.M and Aarts E.H.L., (1987) *Simulated Annealing: Theory and Applications*, Reidel, The Netherlands, 1987.