Applications Performance on Reconfigurable Computers

by

Jang Don Kim

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering in Electrical Engineering and
Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 23, 1997

Copyright 1997 Jang Don Kim. All Rights Reserved.

The author hereby grants to M.I.T. permission to
reproduce distribute publicly paper and electronic copies
of this thesis and grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Prof. Anant Agarwal
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

OCT 29 1997

Applications Performance on Reconfigurable Computers

by

Jang Don Kim

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 1997

In Partial Fulfillment of the Requirements for the Degree of Master
of Engineering in Electrical Engineering and Computer Science

Abstract

Field-programmable gate arrays have made reconfigurable computing a possibility. By placing more responsibility on the compiler, custom hardware constructs can be created to expedite the execution of an application and exploit parallelism in computation. Many reconfigurable computers have been built to show the feasibility of such machines, but applications, and thus compilers required to map those applications to the hardware, have been lacking. RAWCS (Reconfigurable Architecture Workstation Computation Structures) is a methodology for implementing applications on a generic reconfigurable platform.[3] Using RAWCS, two applications, sorting and the travelling salesperson problem, are studied in detail. The hardware constructs and techniques required to map these applications to a reconfigurable computer are presented. These techniques involve the numerical approximation of complex mathematical operations in the simulated annealing solution to the travelling salesperson problem in order to make it possible to map those operations to a reconfigurable platform. It is shown that these integer approximations have no adverse effects on the solution quality or computation time over their floating point counterparts.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science

Acknowledgements

The entire RAW Group contributed significantly to this project. Thanks go to Jonathan Babb for coming up with, creating the framework for, and pushing everyone to work on the RAW benchmark suite. Thanks also go to Matthew Frank and Anant Agarwal for their support, advice, and intoxicating enthusiasm.

I would also like to give thanks to Maya Gokhle who introduced me to the field of reconfigurable computing and helped me to take my first steps.

Finally, thanks to all of my friends and family who never stopped encouraging me.

Table of Contents

1 Introduction ... 13
- 1.1 Field-Programmable Gate Arrays ... 13
- 1.2 Reconfigurable Computers ... 15
- 1.3 Overview of This Thesis ... 18

2 RAWCS ... 21
- 2.1 Introduction ... 21
- 2.2 Compilation Methodology ... 21
- 2.3 Target Hardware ... 23

3 Sorting ... 25
- 3.1 Problem Description ... 25
- 3.2 Algorithms to Find a Solution ... 25
- 3.3 RAWCS Implementation ... 26

4 Travelling Salesperson Problem ... 31
- 4.1 Problem Description ... 31
- 4.2 Algorithms for Finding a Solution ... 31
- 4.3 RAWCS Implementation ... 33

5 Numerical Approximations for Simulated Annealing ... 39
- 5.1 Numerical Approximations ... 39
- 5.2 Data Sets ... 40
- 5.3 Base Case ... 40
- 5.4 Linear Temperature Schedule ... 41
- 5.5 Probability Approximation ... 47
- 5.6 Finding the Best Numerical Approximation ... 49

6 Conclusions ... 55
- 6.1 Merge Sort ... 55
- 6.2 TSP ... 56

Appendix A Merge Sort Source ... 59
- A.1 driver.c ... 59
- A.2 library.v ... 63
- A.3 generate.c ... 66

Appendix B TSP Source ... 71
- B.1 driver.c ... 71
- B.2 library.v ... 82
- B.3 generate.c ... 94

Appendix C Data Sets ... 99
- C.1 80 City TSP ... 99

Appendix D ... 99
- C.2 640 City TSP ... 100

Appendix E Results of Base Case ... 101
- D.1 80 City TSP Solution ... 101
- D.2 640 City TSP Solution ... 102

Bibliography ... 103

List of Figures

Figure 1.1 Typical Setup of a Reconfigurable Computer 15
Figure 3.1 Merge Sort Hardware Construct 28
Figure 3.2 Merge Sort Performance Results 29
Figure 4.1 Parallelizing Energy Change Calculation 37
Figure 5.1 Temperature Cooling Schedule for Base Case 41
Figure 5.2 Performance of Base Case on Two Problems 42
Figure 5.3 Linear Cooling Schedules 43
Figure 5.4 Performance of Linear Cooling Schedules 44
Figure 5.5 Performance of Best Linear Schedule 46
Figure 5.6 Performance of Exponential Approximation 48
Figure 5.7 Performance of Best Approximation 50
Figure 5.8 Performance of Best Numerical Approximation 51
Figure 5.9 Improved Computation Time 53
Figure 6.1 Parallel Implementation of Exponential Approximation 57
Figure C.1: 80 City TSP 99
Figure C.2: 640 City TSP 100
Figure D.1: Solution for 80 City TSP 101
Figure D.2: Solution for 640 City TSP 102

List of Tables

Table 3.1 Merge Sort Performance Results 29

Chapter 1

Introduction

1.1 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) were introduced by Xilinx Corp. in 1986. FPGAs are a class of integrated circuits that is capable of emulating any digital circuit within physical limitations. A digital circuit is specified in software and is then translated into a bitstream in a process known as *synthesis*. This bitstream is then downloaded into the FPGA which gives the chip the necessary information to configure itself to emulate the original digital circuit.

Hardware

The FPGA is usually composed of three kinds of building blocks: configurable logic blocks (CLBs), input/output blocks (IOBs), and interconnection networks.[20] A CLB can be configured to perform any Boolean function of a given number of inputs. IOBs are arrayed along the edge of the chip and are used to communicate from the interior of the chip to the pins on the outside of the chip. The interconnection networks serve two main functions within the FPGA: to connect the I/O blocks to the CLBS and to connect the CLBs to each other.

Synthesis

There are two levels at which a circuit can be described for synthesis. At the highest level, a *behavioral* description in a behavioral Hardware Description Language (HDL) specifies the behavior of a circuit with no implementation details. The logic components and interconnect required to realize the behavior of the circuit are determined by the synthesis tools in a process called *behavioral synthesis*. The output of behavioral synthesis

is a *structural* description of the circuit generally called a *netlist*. Behavioral synthesis is a very complex problem and is not yet well understood. A circuit designer can circumvent the behavioral synthesis stage by describing the target circuit in structural form. This reduces synthesis time and avoids the behavioral synthesis tool which is prone to producing the incorrect structural description. Many HDLs today, such as Verilog and VHDL, offer tools that can translate circuit descriptions that utilize both structural and behavioral constructs giving the designer more flexibility.[24][25]

Logic synthesis is a series of steps that translates the structural netlist to the target bitstream. In the first step, the translator converts the structural netlist to a format understood by the mapping software. The technology mapper then replaces the components in the translated netlist with FPGA-specific primitives and macros. It analyzes timing behavior using the FPGA-specific components to ensure the mapping does not violate the timing constraints of the original design. The partitioner then divides up the mapped netlist among the FPGA chips available on the target system while maximizing gate utilization and minimizing inter-chip communication. The global placer maps each partition into individual FPGAs while minimizing system communication. The global router determines inter-chip routing, and the FPGA APR (automated place-and-route) converts the individual FPGA netlists to configuration bitstreams which are then downloaded into the FPGA.[6]

Applications

FPGAs are used in two broad classes of applications: logic emulation[10] and reconfigurable computing. In logic emulation, a system of FPGAs is used to emulate a digital circuit before fabrication. The FPGA emulation system can be directly integrated into the system under development, and hardware simulations can be performed, albeit at

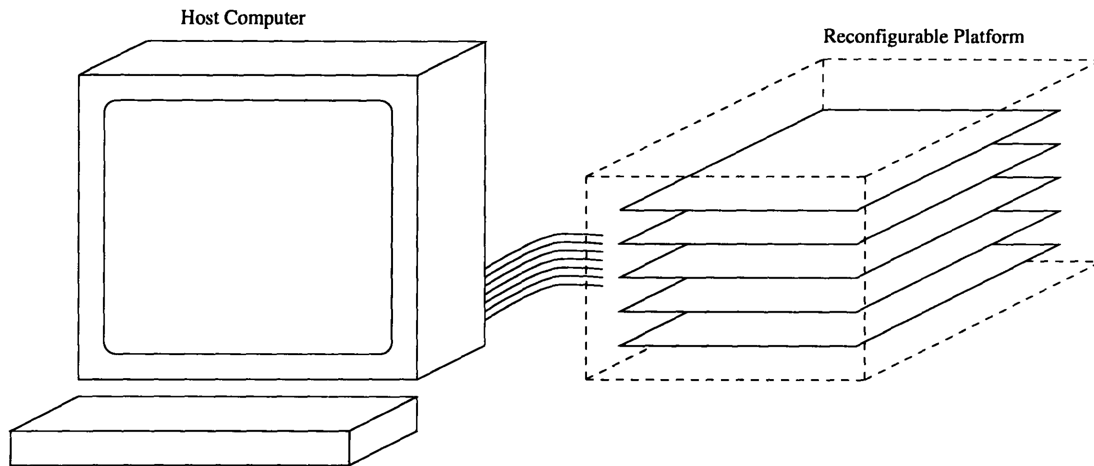


Figure 1.1 Typical Setup of a Reconfigurable Computer

much slower speeds, in order to determine correct behavior of the circuit being designed. Hardware simulations are more reliable and much faster than software simulations.

FPGAs have made also made reconfigurable computing a possibility. Many research groups, mostly in academe, have created hardware platforms and compilers for reconfigurable systems. The basic idea is to execute a program written in a high-level language using application-specific custom hardware implemented in FPGA technology to realize speedups not possible with static general-purpose computers.

1.2 Reconfigurable Computers

A reconfigurable computer refers to a machine that performs all of the same functions as a traditional computer but one in which some or all of the underlying hardware can be reconfigured at the gate level in order to increase the performance of and exploit available parallelism in the application being executed. Figure 1.1 shows the typical configuration of a reconfigurable computer. Generally, these machines have a host processor which controls an FPGA-based reconfigurable platform upon which some part of the application is executed. The reconfigurable platform is used to implement, in custom hardware, operations which do not perform well on a traditional processor but map well to a custom

logic implementation.

Hardware

The hardware of reconfigurable systems can be characterized by four factors:

1. Host interface.
2. Overall system capacity.
3. Communication bandwidth among FPGA resources.
4. Support hardware.

All reconfigurable systems are, to some degree, controlled by a host processor. The level of that control can range from simply providing the configuration bitstream and displaying results to exercising explicit instruction-level control at each clock tick. Accordingly, the interface hardware between the host and reconfigurable system will vary depending upon the nature of the relationship between the two resources.

Overall system capacity refers to the number of gates that the system is capable of emulating. Some reconfigurable systems rely on the multiplexing of a few FPGAs while others have millions of gates of capacity.

The individual FPGAs in the system must communicate. The communication bandwidth among the FPGAs is a major determinant of system performance.

Support hardware refers to any logic that is not an FPGA, specifically memory. Many systems incorporate memory and other devices to augment the capability of the FPGAs.

Compilers

Ideally, a compiler targeted for a reconfigurable computer would allow a user to write a program in a high-level language and automatically compile that application to hardware with no intervention. Of course this ideal is still very far off, and although compiler work in this field has been scant, compilers targeted for these systems can be characterized by two factors: the programming model and the computation model.

Programming Model

The programming model refers to how the user views the relationship between the host and the reconfigurable resource. There are several programming levels at which the user can view the reconfigurable hardware:

1. The user can place the entire application on the FPGA resource relying on the host for configuration bitstreams, data, and a place to return the result.
2. Only certain subroutines may be synthesized so that specified procedure calls from the host will activate the hardware.
3. Lines of code may be converted to hardware which is similar to subroutines but instead of procedure calls, execution of specific lines of code will cause a call to the reconfigurable hardware.
4. Augmentations to the instruction set may be made.

Computation Model

The computation model refers to how the compiler configures the reconfigurable hardware in order to carry out the computations dictated by the programming model. Computation models include the following:

1. Sequential.
2. Systolic array.
3. SIMD array.
4. Hardware generation.[6]

Examples of Reconfigurable Computers

1. Wazlowski *et al* developed PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis) and implemented several small functions such as bit reversal and Hamming distance.[27]
2. Iseli and Sanchez developed Spyder and implemented the game of life.[20]
3. Wirthlin *et al* developed the Nano Processor and implemented a controller for a multi-media sound card.[28]
4. Bertin and Touati developed PAM (Programmable Active Memories) and implemented a stereo vision system, a 3-D heat equation solver, and a sound synthesizer.[7]
5. DeHon developed DPGA-Coupled Microprocessors which are intended to replace traditional processors.[12]
6. Gohkle *et al* implemented dbC (Data-Parallel Bit-Serial C) on SPLASH-2 and NAPA (National Semiconductor's Adaptive Processing Architecture).

[1][2][8][13][14][15][16][17]

7. Agarwal *et al* have developed the VirtuaLogic System and the RAWCS library and have implemented a variety of benchmark programs including sorting and jacobi.[3][26]

1.3 Overview of This Thesis

The goal of this research is to provide insight into what a compiler targeted for a reconfigurable computer must be able to do. This insight will be gained by implementing two non-trivial applications on a reconfigurable computer with a great deal of user involvement in order to better understand the process of going from a high-level programming language description of an algorithm to a custom hardware implementation.

Chapter 2 describes a framework, called RAWCS (Reconfigurable Architecture Workstation Computation Structures), for implementing applications on a generic reconfigurable system. RAWCS provides the user with a methodology for implementing a program written in a high-level language, C, to a custom-hardware implementation of that program on a reconfigurable computer.

Chapter 3 introduces the sorting problem which is the first application studied using RAWCS. The specific algorithm implemented is merge sort. The entire algorithm has been implemented using RAWCS, and performance measures for several sorting problem sizes have been taken. Analysis of those results are presented.

Chapter 4 introduces the travelling salesperson problem which is the second application studied using RAWCS. A framework to implement the simulated annealing algorithm for the travelling salesperson problem has been written, but it has not been compiled or simulated. Modifications to the basic framework that would allow the exploitation of coarse-grained parallelism are presented.

While developing the basic framework, it became apparent that two of the mathematical functions integral to the inner loop of the simulated annealing algorithm

would be very difficult to implement in reconfigurable hardware because of their reliance on exponential math and floating-point operations. Chapter 5 describes a series of experiments that were conducted to study the effects of using a numerical approximation for the calculation of the exponential and using only integer values for all data on solution quality. It has been found that these modifications to the algorithm do not have a noticeable effect on solution quality. Modifications to the basic framework in the form of hardware constructs are given which would allow the exploitation of fine-grained parallelism as a result of the approximation study.

Chapter 6 presents conclusions for these two applications and how these results can be extrapolated to a future compiler for a reconfigurable computer.

Chapter 2

RAWCS

2.1 Introduction

The goal of the MIT Reconfigurable Architecture Workstation (RAW) project is to provide the performance gains of reconfigurable computing in a traditional software environment. To a user, a RAW machine will behave exactly like a traditional workstation in that all of the same applications and development tools will be available on a traditional workstation will also run on a RAW workstation. However, the RAW machine will exhibit significant performance gains brought about by the use of the reconfigurable hardware while at the same time making that use of reconfigurable resources transparent to the user.

RAW Computation Structures (RAWCS) is a framework for implementing applications on a general reconfigurable computing architecture. RAWCS was developed to study how reconfigurable resources can be used to implement common benchmark applications. By studying the implementation of applications on a generic reconfigurable platform, insight into the design and programming of RAW will be gleaned.[3]

2.2 Compilation Methodology

The compilation methodology describes how, given a program in C, RAWCS is used to implement the inner loop of that program in reconfigurable hardware. Although much of the compilation process is performed by the user at this time, the goal is to someday have a similar system completely automated for RAW which will evolve to become the C compiler for RAW.

Programming Model

RAWCS implements the inner loop of applications in reconfigurable hardware and

attempts to exploit as much coarse-grained and fine-grained parallelism as possible through the use of application-specific custom computation structures. It treats the reconfigurable hardware as an abstract sea-of-gates with no assumptions of the hardware configuration.

A program executing on a host machine feeds data to, drives the computation on, and reads the results from the reconfigurable resource. The inner loop can be thought of as a subroutine which is executed on the hardware with all other necessary computations, such as control flow, being executed on the host.

Computation Model

RAWCS takes a directed graph representing the data dependencies of the inner loop of an application and creates specialized user-specified hardware structures. Using the edges of the directed graph, the communication channels among those structures and the interface with the host are also generated.

Compilation Process

The compilation process is as follows:

1. Determine the inner loop to implement in hardware.
2. Design a hardware implementation of the inner loop keeping in mind that creating a parallel structure that uses many copies of the same, small computing elements will ease the latter synthesis stages.
3. Design a hardware library of user-specified computing elements that implement the inner loop of the application.
4. Write a generator program, in C, that takes parameters as inputs and generates a top level behavioral verilog program that 1) instantiates all computing elements and 2) synthesizes the local communication.
5. Write a driver program, in C that is capable of driving either software, simulation, or hardware.
6. Run the software version of the application using the driver program in software mode.
7. Create simulation test vectors using the driver program in simulation mode.

```
8. Simulate the un-mapped design using the Cadence
   verilog simulator.
9. Using Synopsys, synthesize the design.
10.Simulate the synthesized design using the Cadence
   verilog simulator.
11.Compile the design to FPGAs.
12.Download the resulting configuring onto the
   reconfigurable platform.
13.Run the hardware using the driver program in hardware
   mode.
```

2.3 Target Hardware

RAWCS makes no assumptions about the underlying hardware. Initially, RAWCS has been targeted for a generic system composed of a logic emulation system controlled by a host workstation.

VirtuaLogic Emulation System

The VirtuaLogic Emulation System (VLE) from IKOS Systems is a reconfigurable platform on which the RAWCS methodology has been implemented. The VLE consists of five boards of 64 Xilinx 4013 FPGAs each. Each FPGA is connected to its nearest neighbors and to more distant FPGAs by longer connections. The boards are connected together with multiplexed I/Os and also have separate, extensive external I/O.[19]

The VLE is connected to a Sun SPARCstation host via a SLIC S-bus interlace card. A portion of the external I/O from the boards is dedicated to the interface card which provides an asynchronous bus with 24 bits of address and 32 bits of data which may be read and written directly via memory-mapped I/O to the S-bus. The interface is clocked at 0.25MHz for reads and 0.5MHz for writes given a 1 MHz emulation clock. This provides 1-2 Mbytes/sec I/O between the host and the VLE which corresponds to one 32 bit read/write every 100/50 cycles of the 50MHz host CPU.[11]

Virtual Wires

Virtual wires is a compilation technique that overcome pin limitations by multiplexing

physical FPGA pins among multiple logic wires and pipelining these connections at the maximum clocking frequency of the FPGA.

The emulation clock period is the true clock period of the design being emulated. The emulation clock is divided up into phases corresponding to the degree of multiplexing required to wire the design. During each phase, there is an execution part, where the result is calculated, followed by a communication part, where that result is transmitted to another FPGA. The pipeline clock, which is equal to the maximum clocking frequency of the FPGA, controls these phases.

The Softwire Compiler is the VirtuaLogic compiler that synthesizes the bitstream from a netlist for the Emulator. It utilizes both Virtual Wires and third-party FPGA synthesis tools to create the bitstreams.[5]

Chapter 3

Sorting

3.1 Problem Description

The sorting problem can be stated as follows:

> **Input:** A sequence of n numbers $<a_1, a_2, a_3, ..., a_n>$.
> **Output:** A permutation of the input sequence such that $\langle a'_1 \leq a'_2 \leq \ldots \leq a'_n \rangle$. [9]

The data that is being sorted, called the *key*, is generally part of a larger data structure called a *record* which consists of the *key* and other *satellite* data.

3.2 Algorithms to Find a Solution

There are many algorithms to solve the sorting problem. Described here are two common ones used in practice, quicksort and merge sort.

Quicksort

Quicksort uses a divide-and-conquer paradigm as follows:

> **Divide:** The array A{p...r] is partitioned into two subarrays A[p...q] and A[q+1...r] such that each element of the first subarray is larger than each element of the second subarray. The index q is computed as part of this partitioning procedure and is usually the first element in the array.
> **Conquer:** The two subarrays are sorted by recursive calls to quicksort. [9]

Quicksort runs in place meaning that it does not require more memory beyond that required to store the input array in order to compute a solution.

The worst-case running time of this algorithm occurs when the partitioning of the array returns n-1 elements in one subarray and 1 element in the other subarray at each recursive call to quicksort. This results in $O(n^2)$ running time. The best-case running time

of this algorithm occurs when the partitioning of the array returns two subarrays with the same number of elements. This balanced partitioning at each recursive call to quicksort results in $O(nlgn)$ running time. The average-case running time is generally much closer to the best case than the worst case. The analysis is not shown here, but for any split of constant proportionality between the two resulting subarrays for each recursive call to quicksort, the running time will asymptotically approach the best case.

Merge Sort

Merge sort also uses a divide-and-conquer paradigm as follows:

Divide: The array A{p...r] is partitioned into two subarrays A[p...r/2] and A[r/2...r].

Conquer: The two subarrays are sorted by recursive calls to merge sort

Combine: The two sorted subarrays are merged by popping the smaller value at the head of each subarray and storing it in another array until all elements have been popped from the two subarrays. The resulting array is a sorted array with all of the elements from the original two subarrays.

The running time of merge sort is $O(nlgn)$, but it requires additional memory to store a copy of the array during the merge process.

3.3 RAWCS Implementation

Merge sort was chosen as a good candidate to implement on RAWCS because it guaranteed that each recursive call to the procedure would result in two subarrays of the same length. This would guarantee a balanced computation across all of the computation structures.

Also, although merge sort does not compute in place, the algorithm is completely deterministic in the amount of memory that it does require which can be computed at compile time and incorporated into the computation structures.

Software Design

The software to implement the merge sort is written in C and was compiled and executed on a SPARCstation 20 with 192 MB of memory.

The software is executed with a parameter that indicates the number of elements to be sorted. An array of numbers in descending order is generated with that number of elements. This array is then passed on to the merge sort procedure which returns the array in ascending order.

The merge sort uses an iterative procedure to implement the recursive nature of the algorithm. In the first iteration, a merge is done on every pair of elements in the input array by comparing and copying to an intermediate array of the same size. The intermediate array is copied back into the space for the input array after the iteration is completed. During the second iteration, every pair of two elements is merged in the same fashion. This continues to repeat until the pair being merged are exactly the two halves of the input array.

Hardware Design

The hardware structure that implements the merge sort is a binary tree of comparator elements of depth log N, N being the number of elements to compare. The leaves of the tree each contain a single element of the data set that is to be sorted. Each node of the tree performs a comparison of its two inputs from its subnodes, stores the higher value in a register which is connected to the output of the node, and then tells the subnode that had the higher value to load a new value into its register. The subnode that gets the load signal performs the same operation, and so on, until a leaf node is hit. After log N cycles, the largest value is available from the top node, and another value is available on each subsequent clock cycle until the complete sorted list has been output. Figure 3.1 shows a graphical representation of the merge sort hardware construct.

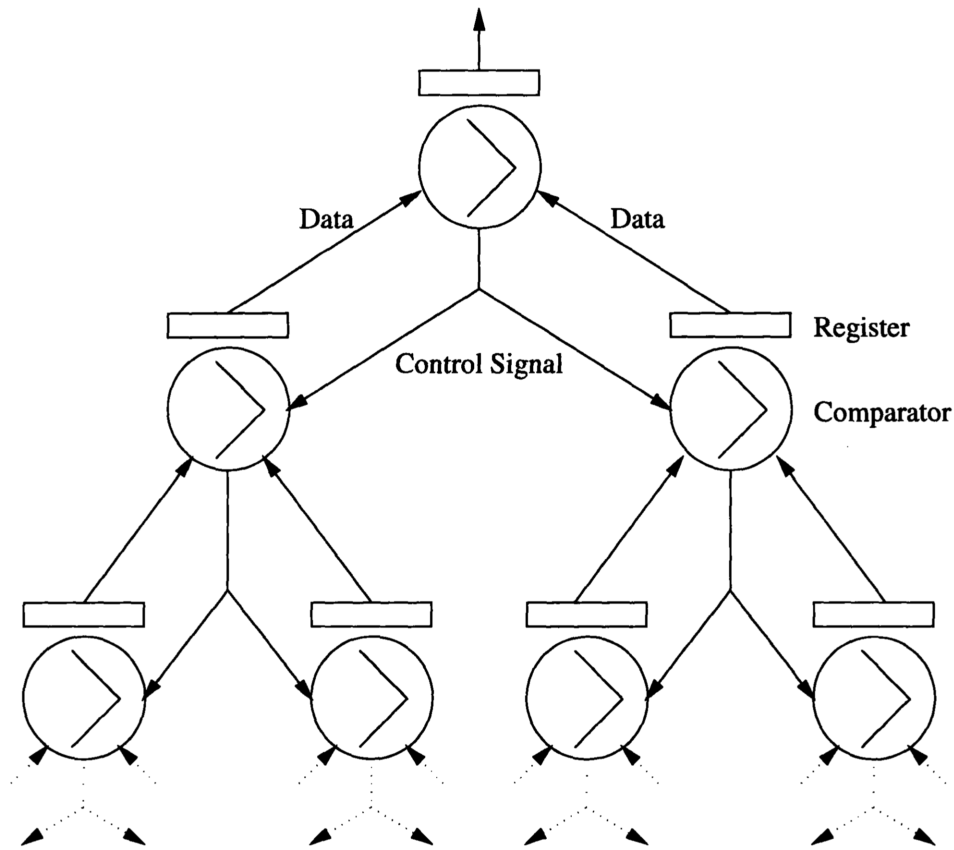


Figure 3.1 Merge Sort Hardware Construct

Performance Results

The data presented here assumes that the internal virtual wires clock is 25MHz. Also, any cycles devoted to I/O between the host and the emulation system is also not included in the results. Only cycles dedicated to actual computation are included in the performance measures. In other words, the cycles required to load data into and to read results from the computation structures are not included.

Results

Unfortunately, the merge sort on RAWCS exhibits only a constant speedup over the sequential C version and does not scale well to larger problem sizes. Table 3.1 shows the speedup that the RAWCS implementation of merge sort has over the sequential C version

for problem sizes of 4, 8, 64, and 256 elements. Notice that the speedup is substantial for the smallest problem size but seems to asymptotically reach a speedup of only 1.6 as the problem size increases. Figure 3.2 is a graphical representation of the data presented in Table 3.1

Size	Clock Rate (MHz)	Hardware Time (µs)	Software Time (µs)	Speedup vs. Software
4	4.17	1.68	16.67	9.92
8	3.13	3.83	16.67	4.35
64	1.79	39.66	66.67	1.68
256	1.19	222.69	366.65	1.65

Table 3.1 Merge Sort Performance Results

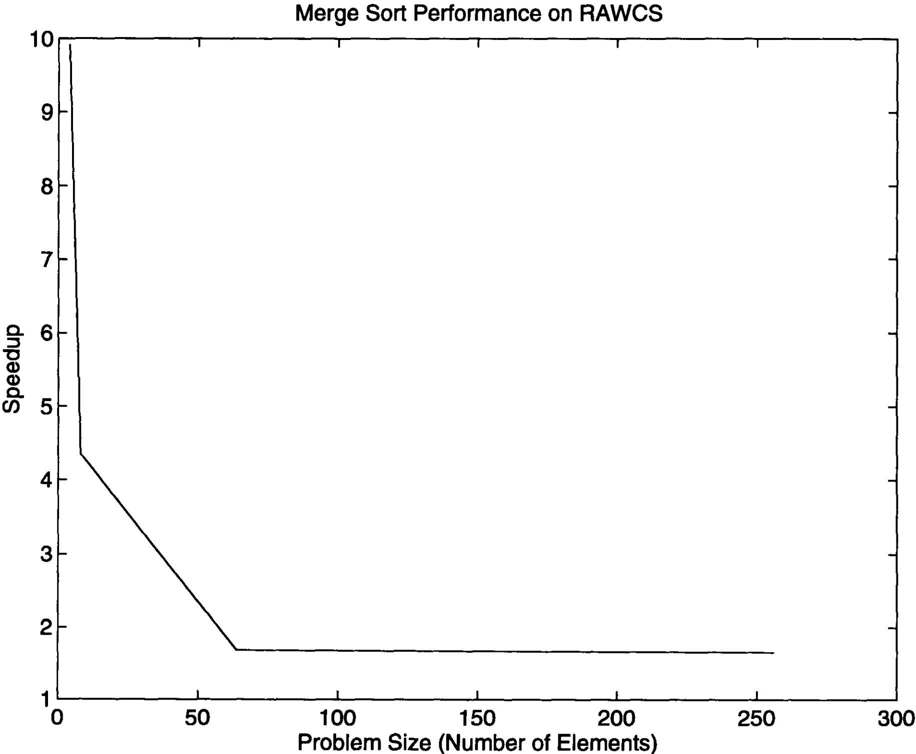


Figure 3.2 Merge Sort Performance Results

Analysis

The disappointing results of the merge sort probably indicate that there is some combinatorial path through the computation structures that is limiting the speed of the computation. This critical path probably comes from the control signals that higher nodes send to lower nodes to load a new value into their register. When the root of the tree sends a load signal, it must propagate all the way down the tree to some leaf node. Since any leaf node could be the final recipient of a root node load signal, every leaf node becomes the terminating end of a critical path. Since the root node and leaf nodes could be separated very far from each other, perhaps even on different boards of the system, this critical path could slow down the clocking rate of the system by a great deal especially as the problem size grows.

One way to alleviate the problem would be to limit the number of nodes in the system to correspond to the number of elements in the problem and to add a new control structure that multiplexed the nodes during the computation. In the current implementation, most of the nodes are idle at any one time, so multiplexing should not pose a significant scheduling problem. By limiting the number of nodes and multiplexing their use, similar to the software implementation, inter-node communication would be greatly reduced and should result in improved performance in both absolute terms and relative to increasing problem sizes.

Chapter 4

Travelling Salesperson Problem

4.1 Problem Description

The travelling salesperson problem (TSP) can informally be stated as:

> A travelling salesperson is given a map of an area with several cities. For any two cities in the map, there is a cost associated which is associated with travelling between these two cities. The travelling salesperson needs to find a tour of minimum cost which passes through all the cities. [18]

Formally, the problem is modeled as an undirected weighted complete graph. There is a path from every city to every other city, and any path can be traversed in either direction. The vertices of the graph represent the individual cities, and the edge weights represent the distances between the cities.

For any cyclic ordered tour through the cities, there is an associated cost function which is simply the sum of the edges between each city pair in the tour. The TSP minimization problem is defined as follows:

> **Instances:** The domain of instances is the set of all undirected weighted complete graphs.
>
> **Solutions:** A feasible solution for a graph is a permutation over its vertices.
>
> **Value:** The value of a solution for the graph is the cost of the corresponding tour. [18]

The TSP is NP-Complete which means that there is no known algorithm that can compute an optimal solution in polynomial time.

4.2 Algorithms for Finding a Solution

The TSP is one of the most studied problems in all of computer science because of its broad applicability to so many different applications such as scheduling and circuit

routing. As such there have been many methods developed to solve or approximate a solution.

Exhaustive Search

A brute force search through all of the possible paths given a graph of cities is, of course, the most simple and the most time consuming. An exhaustive search is guaranteed to find the optimal solution, but the number of paths to search are generally so large that even for very small problems, a large parallel machine is required to actually implement this algorithm. If N is the number of cities in the graph, there are (N-1)! total paths that must be searched. If there are just 50 cities, the corresponding number of paths to search is 6.08E+62!

Branch and Bound

This algorithm continually breaks up the set of all tours into smaller and smaller subsets and calculates a lower bound on the best possible tour that can arise from that subset. An optimal tour is found when there is a path that is shorter than all of the lower bounds of all the remaining subsets. This algorithm can be used to solve TSPs on the order of tens. Going beyond that limit is not actually feasible.[21][22]

Simulated Annealing

Simulated annealing is an approximation algorithm which maps well to solving large TSPs. The algorithm is designed to mimic physical processes where a system at a high energy state is made to come to rest at a desired equilibrium at a lower energy state. For example, consider the physical process of cooling a liquid. In order to achieve a very ordered crystalline structure in the solid state, the liquid must be cooled (annealed) very slowly in order to allow imperfect structures to readjust as energy is removed from the

system. The same principle can be applied to combinatorial minimization problems such as the TSP.[23]

Of course, simulated annealing is an approximation algorithm which means that, although there is a chance that it may find the actual optimal solution, that event is highly unlikely. However, for large TSPs, approximation algorithms are the only feasible solution, and simulated annealing has been shown to provide very good approximations for large problem sizes.

4.3 RAWCS Implementation

Although simulated annealing is only an approximation algorithm, it was chosen as the candidate for RAWCS implementation because of its wide applicability to many different problems. Any kind of optimization problem where an exact answer is not a necessity can utilize this algorithm. Furthermore, there are many opportunities to exploit parallelism in the algorithm as will be discussed later.

Software Design

The software to implement the simulating annealing is written in C and was compiled and executed on a Sun (Sun Microsystems Inc.) SPARCstation LX with 32 MB of memory.

The software assumes that the input is a Euclidean TSP in the plane that satisfies the triangle inequality. This simply means that the software assumes that the problem is a weighted fully connected undirected graph in a single plane where every three cities, A, B and C, satisfies the triangle equality. Satisfying the triangle equality simply means that the distance between any two cities, A and B, is shorter than the sum of the distances from A to B and from B to C.[18]

The software reads in a file containing cities and their respective X,Y coordinates. It then calculates all of the inter-city distances and stores these values in a matrix. After

calculating the minimum spanning tree (MST) for the fully connected, undirected graph, the software uses the MST to find an initial path through the cities. This initial ordering is then passed on to the annealing procedure.

The simulating annealing procedure is a huge loop within a loop. The outer loop is the temperature cooling schedule. When the temperature falls below the final temperature, the annealing ends, and the current tour is returned as a solution. In the inner loop, there are a maximum of 500 * N, N being the number of cities in the graph, iterations performed at each temperature. If there are ever more than 60 * N swaps that are accepted, either by the fact that they shorten the tour or are accepted probabilistically, at any temperature, then the inner loop is broken even though 500 * N swaps have not been tried.

To simulate annealing, the procedure chooses two cities at random, swaps them, and calculates the change energy caused by the swap. This involves only six distance queries into the inter-city distance matrix. If B and E are the two cities chosen at random to be swapped, and A is the previous city to B in the current tour, C is the city after B in the current tour, and similarly for D and F for E, then calculating the energy change involves adding the distances from A to E and from E to C and D to B and B to F (the new tour) and subtracting the distances from A to B and from B to C and D to E and E to F (the previous tour).

Once the energy change for a random swap has been computed, the swap is accepted if the energy change was negative meaning that the new tour is shorter with the swap. If the energy change is positive meaning that the new tour is actually longer with the swap, it is accepted with a certain probability. That probability is determined by calculating the following equation:

$$\text{Random Number} < e^{\frac{-EnergyChange}{Temperature}}$$

When either 500 * N swaps have been attempted or 60 * N swaps have been accepted, the temperature is decreased and the inner loop repeats. The process continues until the temperature falls below the pre-specified final temperature, or there is an iteration where no swaps are accepted.

Hardware Design

The first step in implementing simulated annealing on RAWCS is to identify the inner loop that will be executed on the reconfigurable emulation system. The obvious choice is to place the entire simulated annealing loop on the VirtuaLogic System.

The goal of the first phase of the RAWCS hardware design for simulated annealing was to create computation structures that would implement the simulated annealing loop but would also provide a framework to exploit both coarse-grained and fine-grained parallelism by extending and augmenting that framework.

In this first phase, there are three modules that were created:

1. A **distance matrix** which holds all of the inter-city distances. There is only one such matrix in the current implementation. A single distance request from each simulated annealing module is processed at each clock tick.
2. A **simulated annealing** module which actually performs the annealing. Each module works on its own private copy of a tour. At the end of annealing, the module with the shortest tour returns it as the solution.
3. A **control** module which controls the operation of the simulated annealing modules. It provides the temperature cooling schedule and notifies the simulated annealing modules to return solutions when the temperature cools to below the threshold.

Exploiting Parallelism

Coarse-grained parallelism refers to how the interconnection of the various modules can be altered to exploit parallelism. Fine-grained parallelism refers to how the modules

themselves can be modified to exploit parallelism of the computations made within the modules.

Coarse-Grained

The first modification that could be done would be to have a global current tour that is operated on by all of the simulated annealing modules simultaneously. This would require locking mechanisms on such a global tour data structure so that no two simulated annealing modules will try to swap the same city to two different locations. Having a global tour would allow many more swaps to be attempted during the same execution time.

Another modification that could be made would be to have multiple distance matrices each with multiple ports. Currently, there is only one distance matrix that services a single request per simulated annealing module per clock tick. Having a matrix for each simulated annealing module that can handle six distance requests per clock tick would allow all of the inter-city distance queries to complete in a single tick. In RAWCS, it is a simple matter to create a memory or bank of registers with multiple ports to parallelize the accesses into the distance matrix. It is guaranteed, from the previous modification, that each of the six distance accesses will be unique, so there will be no contention for a memory location. Furthermore, the inter-city distance matrix does not change over time, so multiple simulated annealing modules having their own private multi-ported inter-city distance matrix would not pose any coherence problems.

Fine-Grained

In the main simulated annealing loop, there are three basic computations performed during computation:

1. Once the swap is chosen at random, multiple queries are made into the distance matrix.

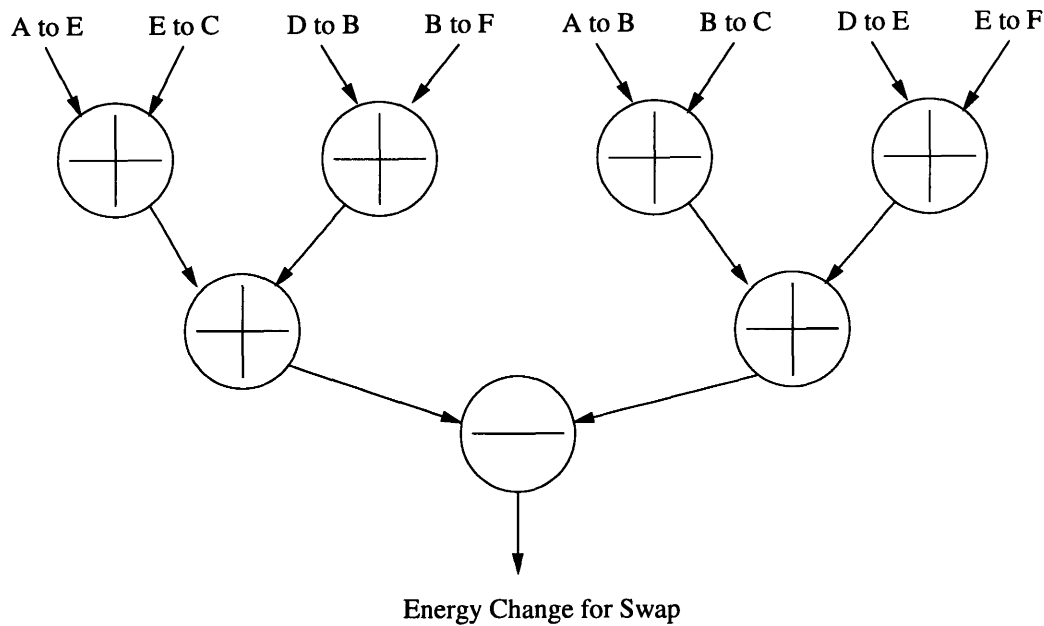


Figure 4.1 Parallelizing Energy Change Calculation

2. The energy change calculation is done based on the results of the queries into the distance matrix.
3. For swaps with positive energy change, calculation of the exponential is performed to see if the swap will be accepted anyway.

The energy change calculation can easily be parallelized in custom hardware to execute in O(log N) time, N being the number of terms in the summation, by using the hardware construct shown in Figure 4.1. The inputs are simply the distances between the respective cities where B and E are the cities to swap, A is the city before and C is the city after B in the current tour and, D is the city before and F is the city after E in the current tour.

Parallelizing the calculation of the exponential in determining acceptance of swaps with positive energy change is not possible. There is no opportunity to parallelize the calculation of an exponential. Furthermore, the use of floating point values in these calculations makes them extremely difficult to map to reconfigurable logic.

Chapter 5

Numerical Approximations for Simulated Annealing

5.1 Numerical Approximations

In order to implement the simulating annealing algorithm on reconfigurable hardware, all of the floating point operations must be converted to equivalent integer computations. This conversion must be done because the floating point operations in the implementation of the algorithm do not map well to reconfigurable hardware, nor do they offer any opportunities to exploit parallelism using custom hardware. This study is intended to answer the following question:

```
What effect does the conversion to integer approximations
  have on the simulated annealing algorithm in terms of
        solution quality and computation time?
```

There are two floating point math operations that are at the heart of the simulated annealing algorithm. The first is the temperature cooling schedule, and the second is the calculation of the exponent in the probabilistic acceptance of a positive energy change.

Generally, the temperature cooling schedule is an exponential function with the initial temperature starting at a high value and decreasing by some fixed percentage at each iteration of the algorithm. To convert this schedule to the integer domain, a linear cooling schedule must be adopted.

A numerical approximation for the exponential function in the probabilistic acceptance of a positive energy change during the annealing is of the following form:

$$X = \frac{-energyChange}{Temperature(T)}$$

$$e^X = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \ldots$$

Given that X is an integer, this approximation can be completely done using integer values.

5.2 Data Sets

The data sets for the simulated annealing were created at random. A random number generator was used to create a random map of cities based on a parameter denoting how many cities the map should have.

Maps for 20, 80, 320, and 640 cities were created. These values were chosen in order to have two TSPs at the low end and two at the high end because it was assumed that conclusions about data sets in the middle could be inferred from the results based on these graphs. Refer to Appendix C for a graphical representation of the data sets.

5.3 Base Case

A base case had to be chosen as a metric to measure performance of all of the variations on the simulated annealing algorithm. The parameters for the base case were 100 for the initial temperature, a cooling rate of 0.9, a final temperature of 0.1, and the use of the exp function in the gcc math.h library to calculate the exponential in the probabilistic acceptance of positive energy changes.

An initial temperature of 100 was chosen because the way the city graphs were created, the energy changes from swaps were generally in the hundreds. Furthermore, at a cooling rate of 0.9, an initial temperature of 100 had 66 distinct temperatures before falling below the final temperature of 0.1. A maximum of 66 iterations seemed reasonable. A cooling rate of 0.9 and a final temperature of 0.1 were chosen for the same reasons.

Results

Figure 5.1 shows the temperature cooling schedule given the parameters for the base case. The graph is an exponential curve.

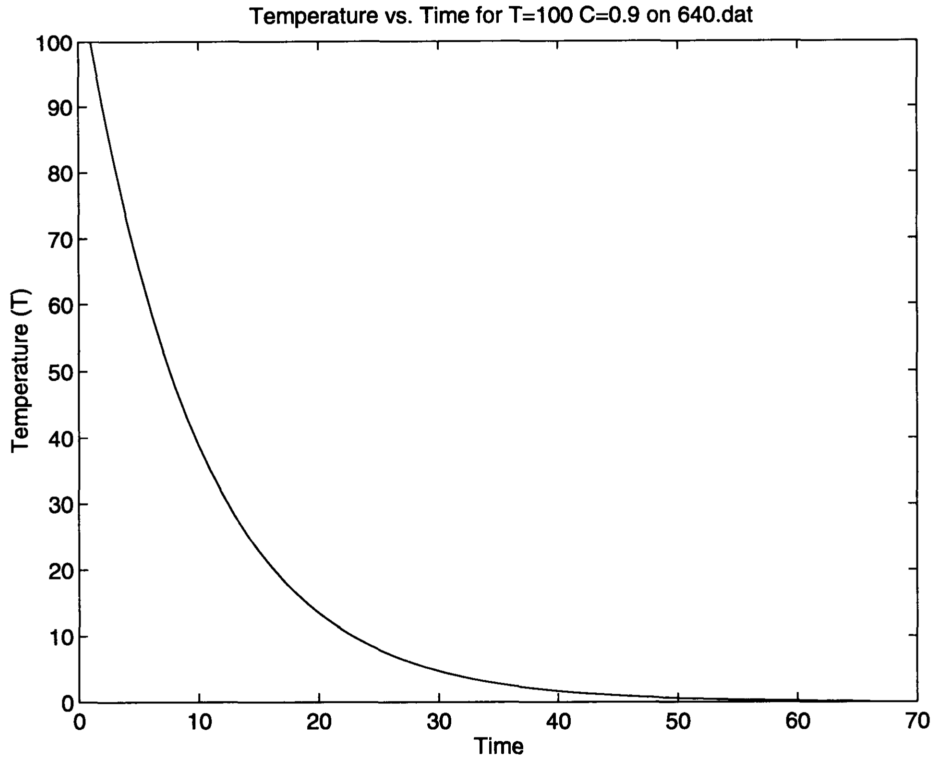


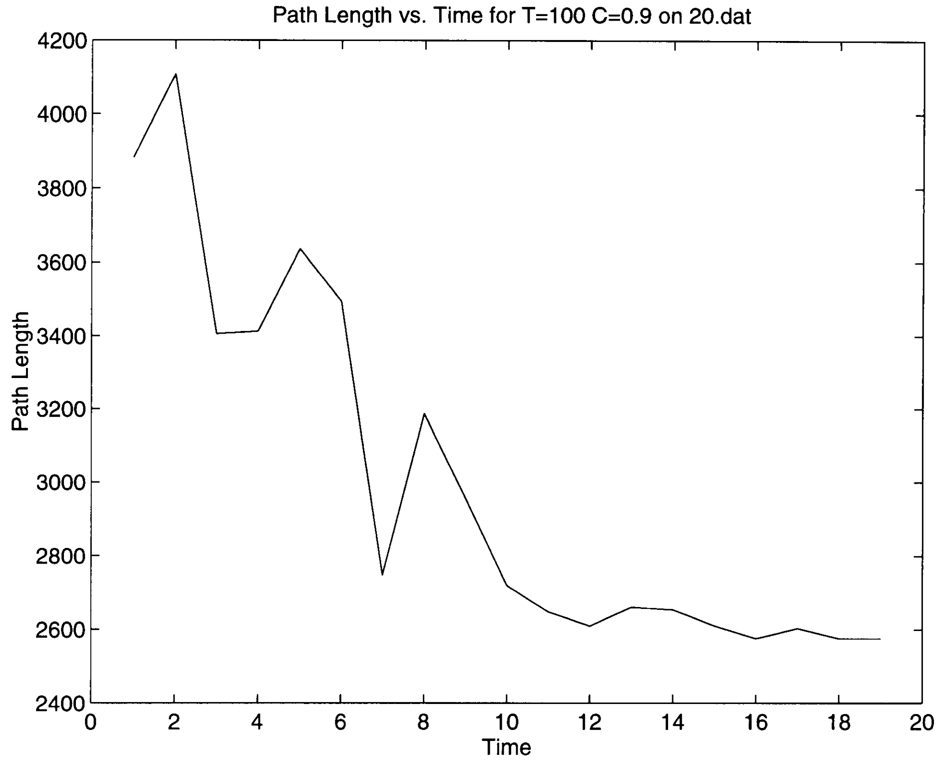
Figure 5.1 Temperature Cooling Schedule for Base Case

Figure 5.2 shows the path length for the current tour as the simulating annealing is performed on the 20 and 640 city TSPs. Notice that the path length is not always decreasing and that there are several points where the path length actually increases meaning that the probability function caused swaps with positive energy changes to be accepted. Notice how closely the path length improvements follow the cooling schedule especially in the 640 city problem.

Refer to Appendix D for a graphical representation of the TSP solutions that the base case found for the 80 and 640 city problems.

5.4 Linear Temperature Schedule

The first variable that was studied was the temperature cooling schedule. What effect does a linear temperature schedule have on the solution quality and time for computation of the simulated annealing?



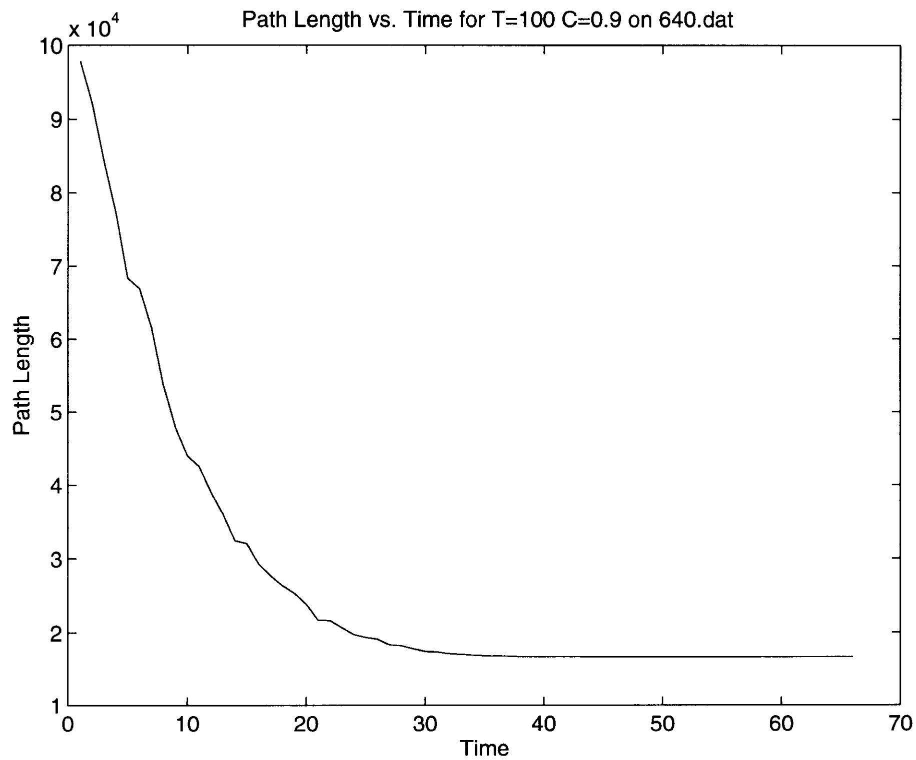


Figure 5.2 Performance of Base Case on Two Problems

The restrictions on these experiments were that the parameters chosen had to ensure that the number of distinct temperatures possible during cooling had to be 66 which is the

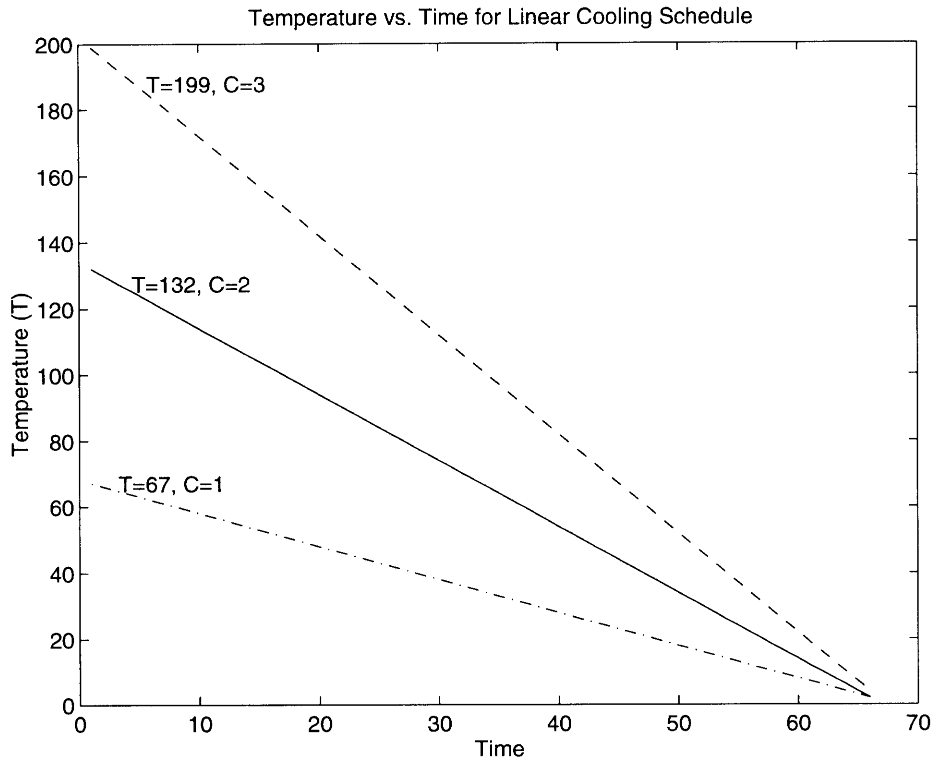


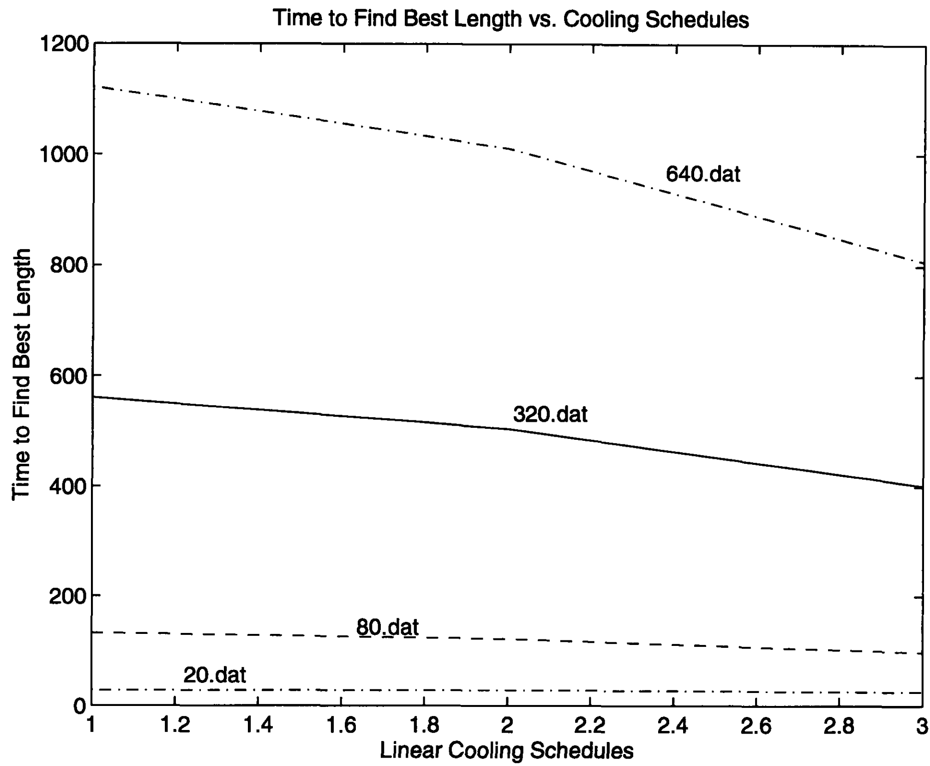
Figure 5.3 Linear Cooling Schedules

same as the base case. This was to ensure that a new cooling schedule would not perform better or worse simply because it had more or less opportunity to find a better solution than the base case. Given that restriction, the initial temperature could also not deviate too far from the initial temperature of 100 in the base case. This was to insure that what was affecting the solution was not the magnitude of the temperature but the slope of the cooling schedule. Also, the gcc exp function was still used to calculate the exponential in the probabilistic acceptance function.

The parameters chosen for these experiments were initial temperatures of 67, 132 and 199 and cooling rates of 1, 2 and 3 respectively. These parameters fit the restrictions. No other set of parameters was really possible while staying within the restrictions.

Results

Figure 5.3 shows the linear cooling schedules for the three sets of parameters.



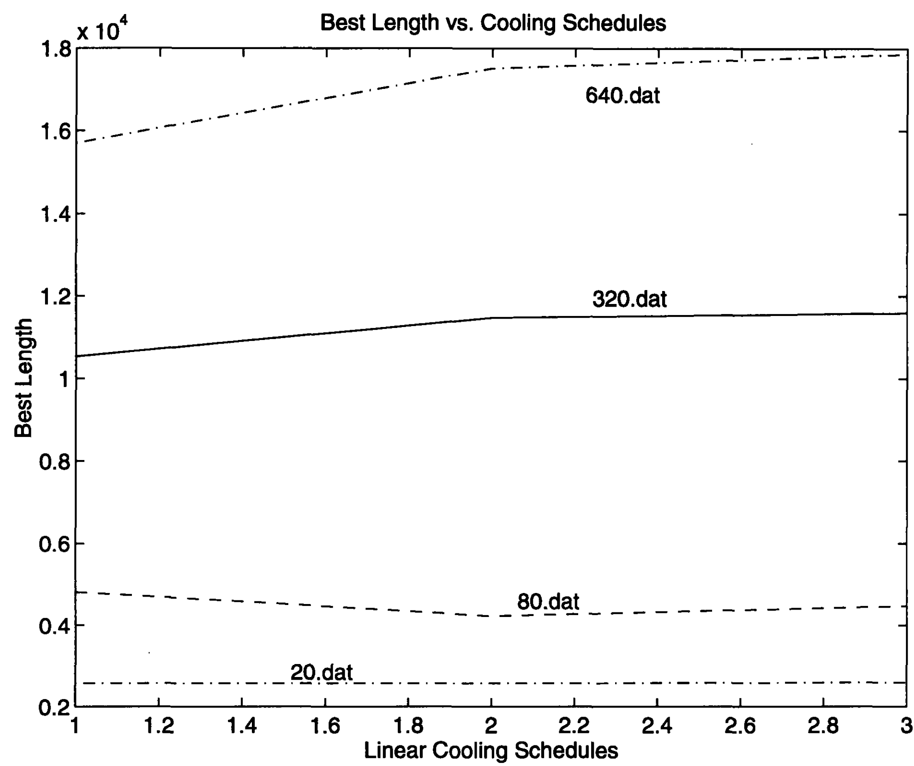


Figure 5.4 Performance of Linear Cooling Schedules

Figure 5.4 shows the performance of the three linear cooling schedules on the four

problem sizes. The top graph shows how long it takes the three different schedules to complete their respective computations on the four randomly generated TSPs. On the y-axis, 1, 2 and 3 refer to initial temperatures of 67, 132 and 199 respectively. Notice that the initial temperature of 67 takes the longest time to compute and that the compute time decreases as you increase the initial temperature.

The bottom graphs shows the length of the optimal tour produced by each linear cooling schedule on the four randomly generated TSPs. Notice that as the initial temperature is increased, the length of the optimal tour found by simulated annealing increases.

An initial temperature of 67 seems to produce the best answer among the three different schedules, albeit at a cost of longer computation time. Figure 5.5 shows the length of the current ordering during annealing of this best cooling schedule. The upper graph shows execution on the 20 city TSP. Notice how much more chaotic the path length is than the equivalent graph for the base case. The lower graph shows execution on the 640 city TSP. Again, notice how closely the path length during execution follows the linear cooling schedule.

Analysis

There is an obvious trade off in time vs. solution quality among the three linear cooling schedules. The lower the initial temperature, the better the solution quality but the longer the computation time. Why does this occur?

The higher the initial temperature, the greater the volatility in the annealing process which would tend to produce poorer results. With an exponential decay, a high initial temperature quickly drops away so that the effects of high temperatures on the probability function for positive energy changes is short-lived. However, with a linear decay, the

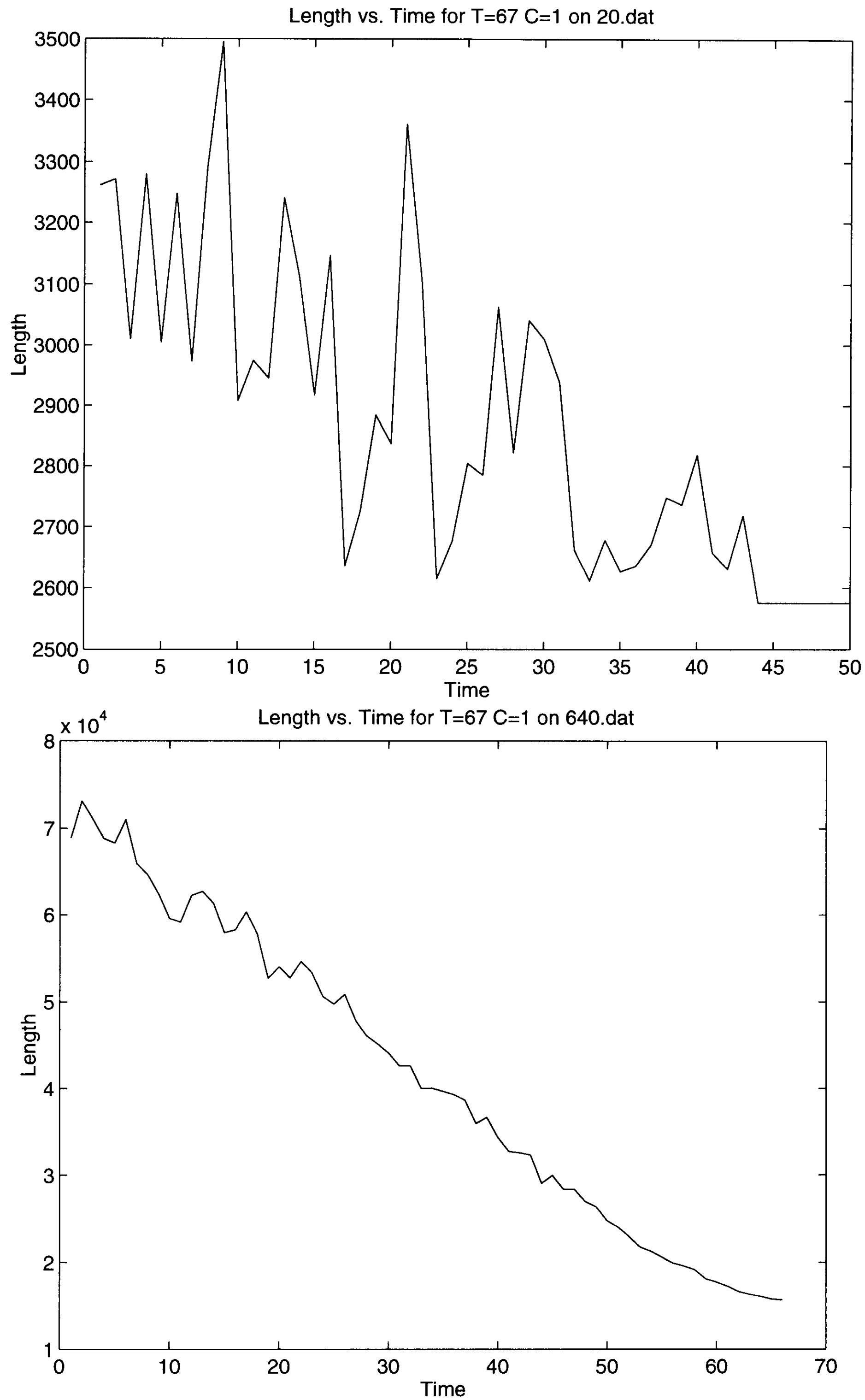


Figure 5.5 Performance of Best Linear Schedule

temperature stays high much longer such that the volatility of the system is much greater. The total potential energy of the system can be equated with the area under the graph of

the cooling schedule, so that high initial temperatures have a magnified effect on the quality of the final solution thus leading to poorer solutions.

Also, at higher initial temperatures more swaps with positive changes in energy are accepted. Since the total number of swaps that are permitted per distinct temperature are limited by 60 * N, N being the number of cities in the problem, the higher initial temperature schedules hit this limit much more frequently for each distinct temperature than the lower temperature schedules. The lower temperature schedules must try many more swaps at each temperature because fewer positive swaps are probabilistically accepted.

5.5 Probability Approximation

The second variable studied was the calculation of the exponential in the probabilistic acceptance of swaps which actually increase the length of the current tour. What effect does an approximation for the exponential have on the solution quality and time for computation of the simulated annealing?

The parameters for these experiments were 1, 2, 3 and 4 terms in the following approximation equation for the exponential:

$$X = \frac{-energyChange}{Temperature(T)}$$

$$e^X = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \ldots$$

1 term means a single X term was used in the approximation. The initial temperature was 100 and the standard exponential cooling schedule of 0.9 was used as in the base case. All other values were integers, and when temperature was used in a calculation, the result was cast to type int before being assigned to the variable.

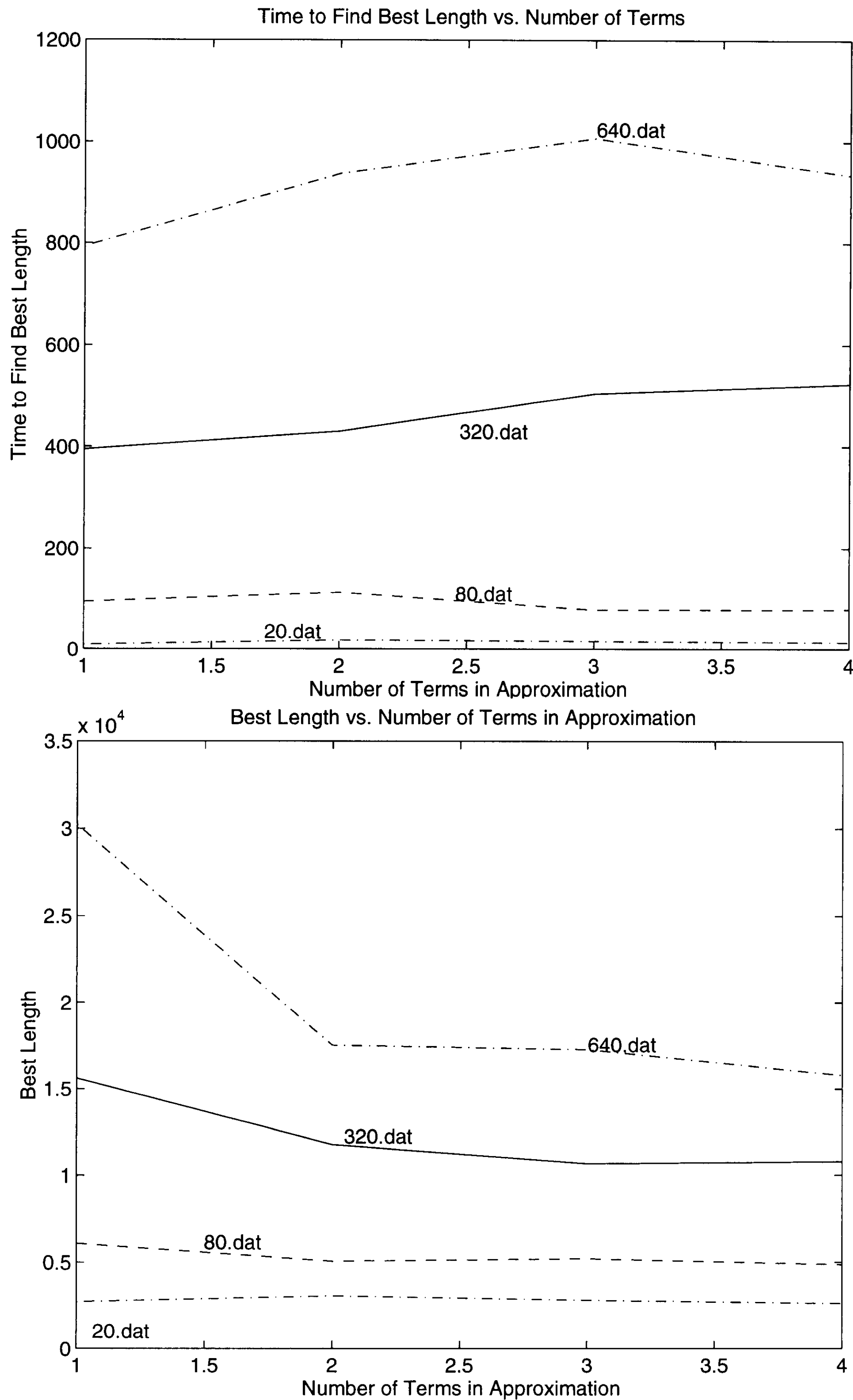


Figure 5.6 Performance of Exponential Approximation

Results

Figure 5.6 shows the performance for the four approximations for the four problem

sizes. The Y-axis indicates the number of terms used in the approximation. The upper graph shows the time for computation. Generally, increasing the number of terms increases the computation time since there are simply more calculations to perform when the swap has positive energy change.

The lower graph shows the length of the shortest tour found by each approximation on the four problem sizes. Notice the huge increase in solution quality, especially for the large TSP, when the approximation goes from 1 term to 2. Unfortunately, this huge jump causes the other improvements to seem small due to the scaling of the Y-axis, but the improvement in solution quality as the number of terms in the approximation is increased is quite appreciable.

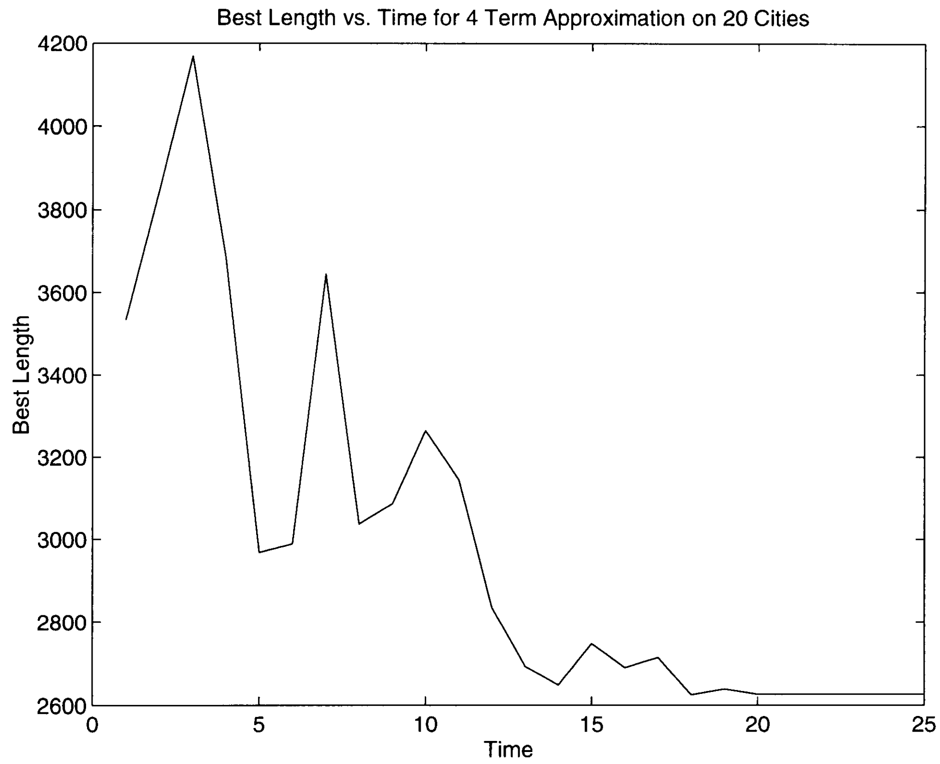
Figure 5.7 shows the length during execution of the 4 term approximation on the 20 and 640 city TSPs. Notice how similar these graphs are to the corresponding graphs for the base case.

Analysis

The 4 term approximation provides the best solution quality but also runs the slowest because it simply has more computations to perform than the approximations with fewer terms. This is not a surprising result.

5.6 Finding the Best Numerical Approximation

The best numerical approximation should arise from combining the linear cooling schedule and the exponential approximation which gave the shortest path tours. The best linear cooling schedule was an initial temperature of 67 and a cooling rate of 1. Four terms in the approximation gave the shortest path.



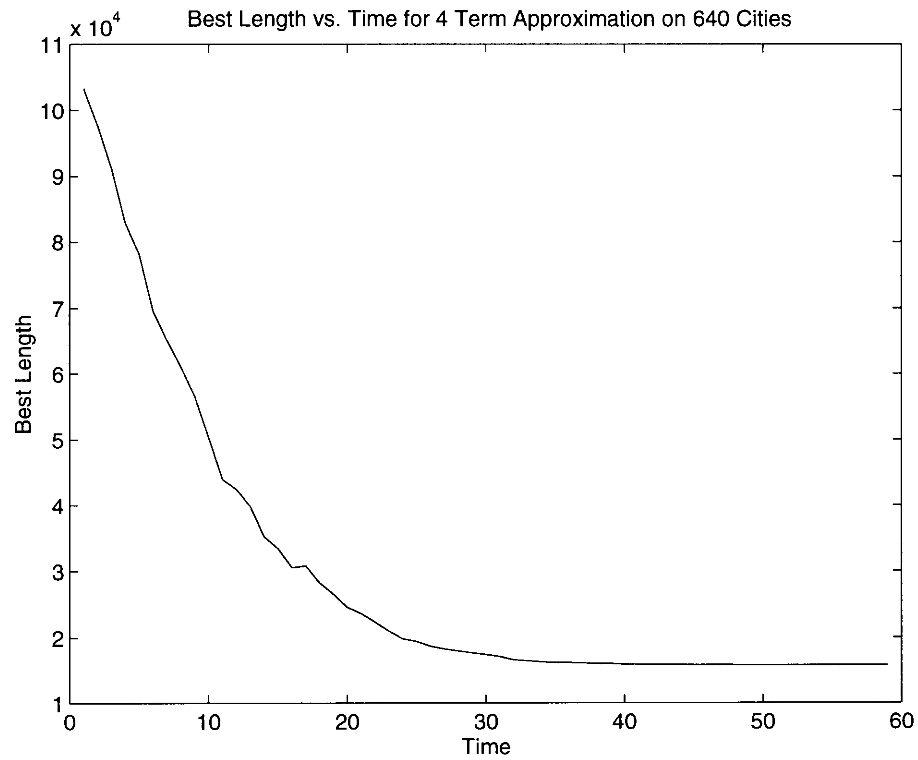


Figure 5.7 Performance of Best Approximation

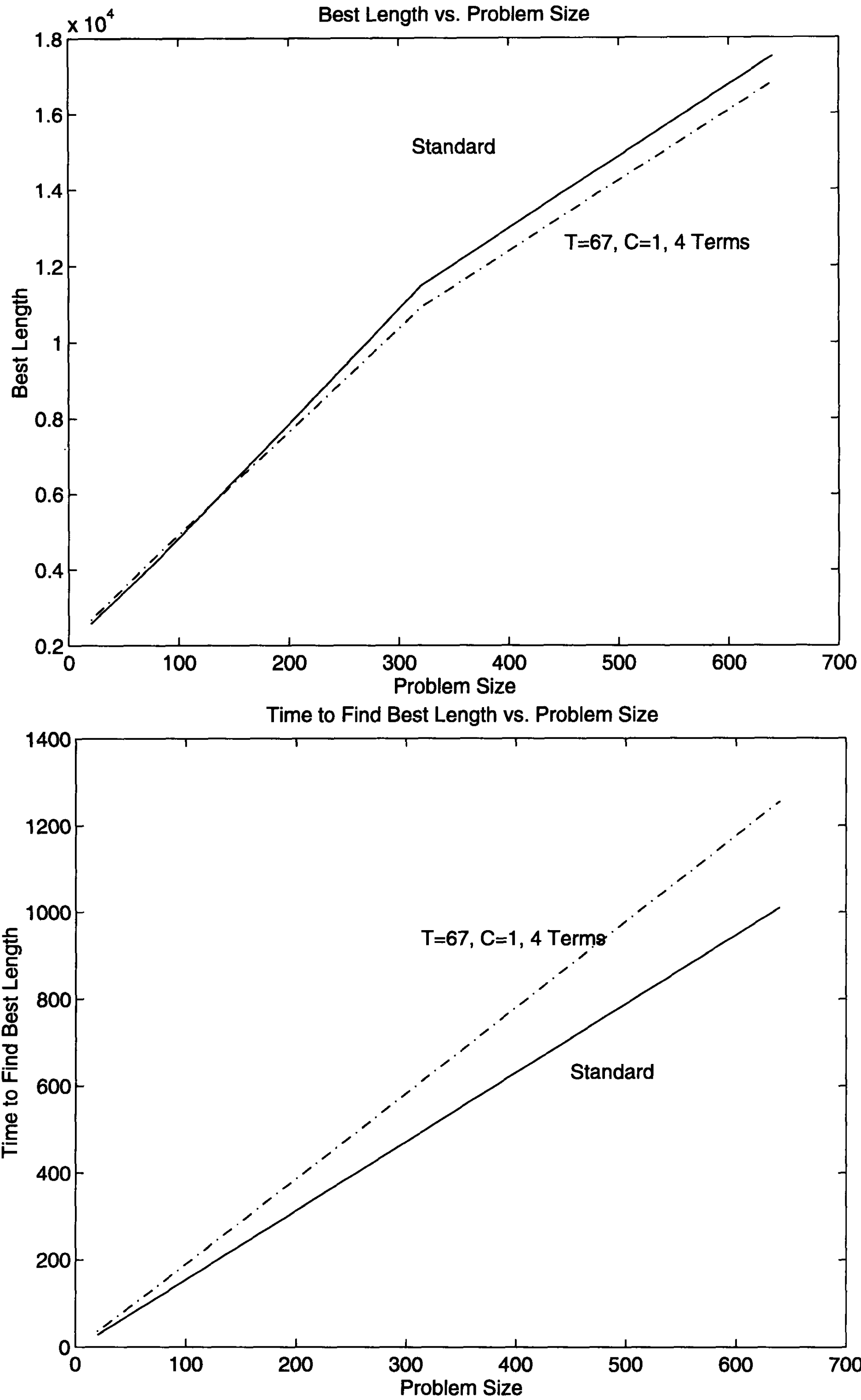


Figure 5.8 Performance of Best Numerical Approximation

Results

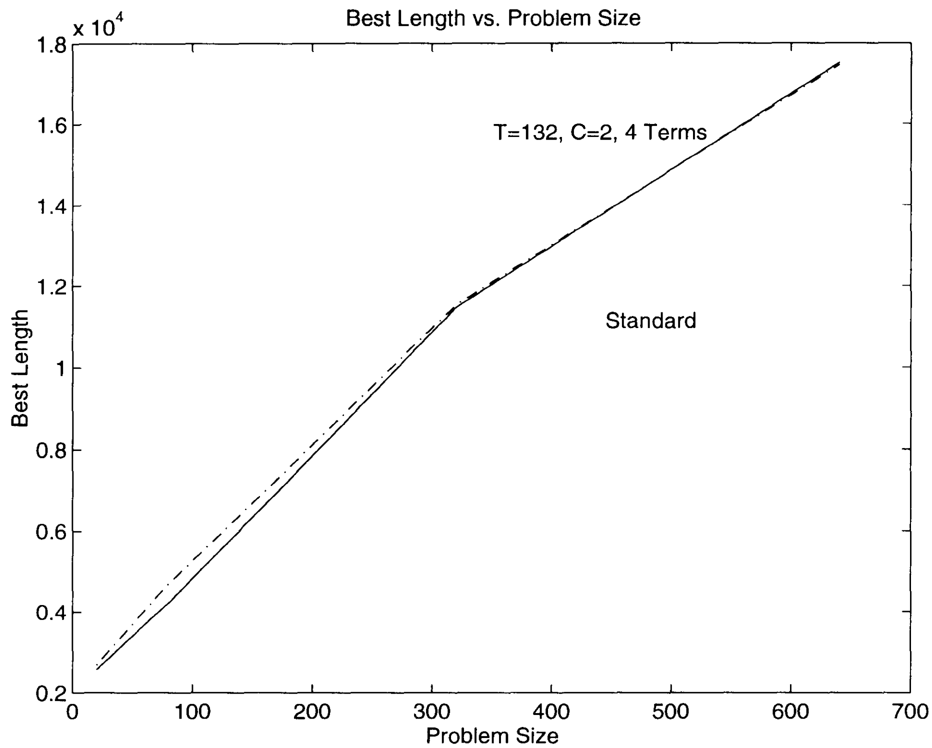
Figure 5.8 shows the shortest path found for all four problem sizes for both the base

case and the complete numerical approximation using an initial temperature of 67, a linear cooling rate of 1, and four terms in the approximation for the exponential. The numerical approximation actually finds better solutions than the base case, but it takes longer to execute. The difference in solution quality grows with the problem size, but so does the computation time. As a matter of fact, the difference in computation time seems to grow faster than the improvements in solution quality.

By increasing the slope of the temperature cooling schedule, there should be a corresponding decline in both solution quality and time for computation. Figure 5.9 shows the shortest path found for all four problem sizes for both the base case and the complete numerical approximation using an initial temperature of 132, a linear cooling rate of 2, and four terms in the approximation for the exponential. The numerical approximation almost exactly mimics the solution quality given by the base case, and it also performs slightly better in terms of computation time.

Analysis

As can be seen from the results, the complete numerical approximations work just as well or better than the floating point base case.



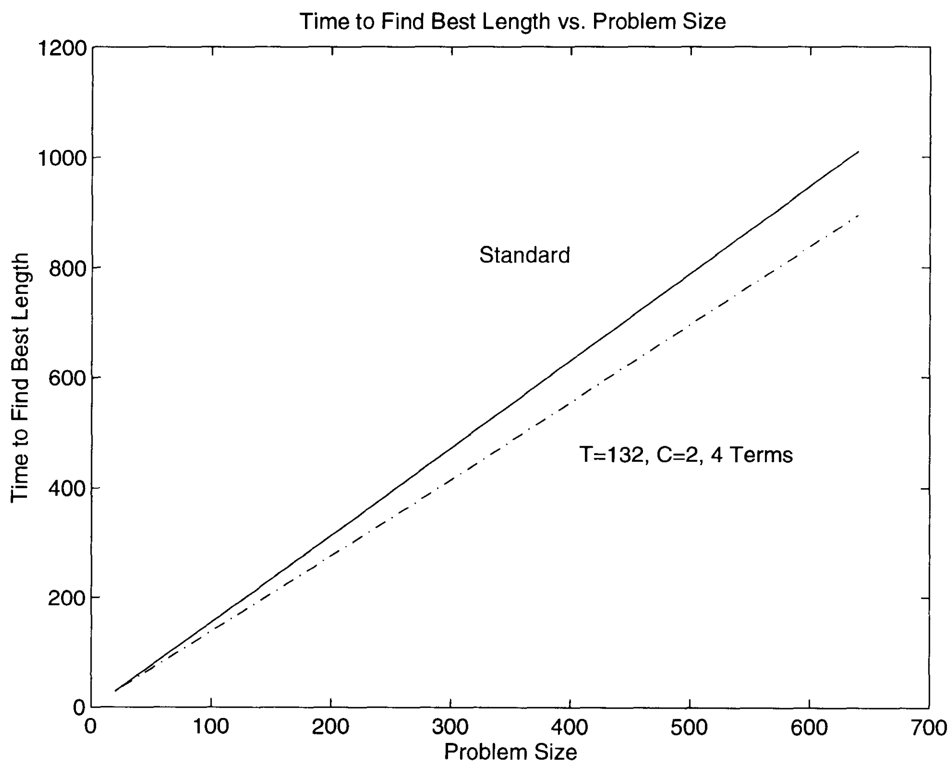


Figure 5.9 Improved Computation Time

Chapter 6

Conclusions

6.1 Merge Sort

The merge sort algorithm has been successfully implemented using RAWCS. Unfortunately, the implementation only exhibits a constant speedup over execution on a traditional workstation. This lack of performance gain is probably due to a critical communication path in the hardware that is slowing the emulation clock rate.

The RAWCS merge sort implementation shows that a compiler for a reconfigurable computer must do several things. First, such a compiler must be able to glean all of the possible parallelism out of the application in order to make the use of reconfigurable resources worthwhile. Even ignoring I/O limitations, reconfigurable hardware will never be clocked as fast as a microprocessor, so exploiting parallelism is the only way to make the use of reconfigurable hardware feasible.

Second, synthesizing for FPGAs is an extremely long and arduous task. Even on the fastest workstations, completely synthesizing a design takes on the order of days to complete. So, as in merge sort, a compiler for a reconfigurable computer must focus on generating hardware modules which can simply be replicated for larger problem sizes in order to cut down on synthesis time. Then, once a single module is synthesized, making copies of that same module simply involves placement and routing without the need to re-synthesize that module. If a compiler generated many unique elements, then the synthesis time, even assuming future improvements in the synthesis software, may negate the performance gains that reconfigurable computers offer.

Finally, a compiler for a reconfigurable computer must take communication requirements into careful consideration when implementing hardware constructs for an

application. The RAWCS merge sort implementation attempted to exploit parallelism without regard to efficient use of the generated hardware or the communication requirements that such a large collection of modules would require. Intelligent multiplexing of hardware must be at the heart of any compiler for a reconfigurable computer in order to minimize inter-module communication and also to minimize reconfigurable hardware utilization which is generally at a premium and limited on a reconfigurable computer.

Refer to Appendix A for the source code for the merge sort implementation using RAWCS.

6.2 TSP

A basic implementation of the simulated annealing algorithm has been written but has not been compiled or simulated. During development, it was discovered that the temperature cooling schedule and the calculation of the exponential in the probabilistic acceptance swaps with positive energy change could not be mapped to reconfigurable hardware.

Therefore, experiments were conducted to study the effects of using only integer values for all data and approximations for the exponential. It was found that these changes had no noticeable effect on the solution quality of the algorithm which means that the inner loop of the simulated annealing solution to the travelling salesperson problem should map well to RAWCS which, in turn, should be able to take full advantage of the fine-grained parallelism in the approximated version. The custom hardware to implement the exponential function in parallel is shown in Figure 6.1. In addition, parallelizing the distance accesses and energy change calculation, as shown before, should give the simulated annealing algorithm good speedup with RAWCS over the sequential implementation.

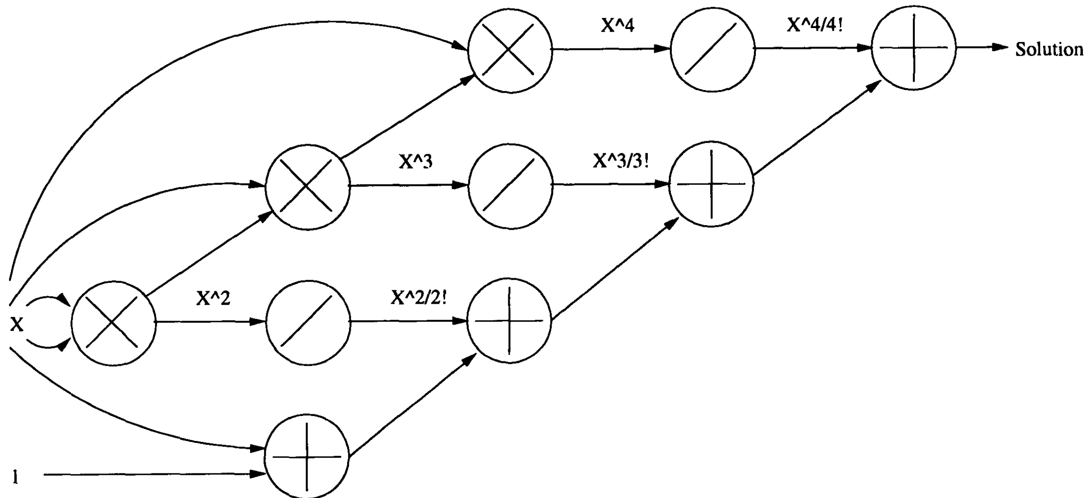


Figure 6.1 Parallel Implementation of Exponential Approximation

The implementation of the simulated annealing algorithm shows that a compiler for a reconfigurable system must be capable of making the same kinds of approximations automatically. Whether it is complex math functions, as in simulated annealing, or some other functions that do not map well to reconfigurable hardware, the compiler must recognize those opportunities where an approximation will suffice instead of throwing up its hands and saying that it is impossible to use the reconfigurable resource to implement a particular problem.

Refer to Appendix B for the source code for the TSP implementation on RAWCS.

The RAW Compiler

The work presented here only touches the tip of the iceberg in terms of the analytical ability and complexity that a compiler for a reconfigurable computer must have. Essentially, a compiler for such a system must automatically do many of the things that an experienced hardware designer must do. It must have the ability to find and create hardware to exploit parallelism in applications keeping in mind hardware efficiency and communication minimization. The compiler must also recognize and formulate

approximations for functions and operations that do not map well to reconfigurable hardware in such a way that those approximations do not degrade solution quality.

As the RAW architecture has developed, the amount of silicon devoted to reconfigurable logic has slowly decreased. A factor in this decline is the complexity and overhead that the greater use of reconfigurable logic would demand of the compiler.[26] It is clear that reconfigurable logic will have a place in future computer systems. However, how large a role reconfigurable logic plays in those systems is not yet clear. There must be a median where performance gains overshadow the compiler overhead and complexity. That median is yet to be found.

Appendix A

Merge Sort Source

A.1 driver.c

```
#include <coarsecoarse/types.h>
#include <generate_verilog.h>
#include <interface.h>
#include <tval.h>

#define TIMER_RPTS 1000
#define SCANID 2000

#ifdef SOFTWARE
print_array(int a[], int size)
{
  int i;

  for (i=0;i<size-1;i++)
    printf("%i, ", a[i]);
  printf("%i\n", a[size-1]);
}
#endif

#ifndef HARDWARE

void merge(int a[], int b[], int c[], int m, int n)
{
  int i=0, j=0, k=0;

  /* Merge the two arrays. */
  while (i < m && j < n)
    if (a[i] > b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];

  /* If array b runs out before a, then fill c with the rest of a. */
  while (i < m)
    c[k++] = a[i++];

  /* If array a runs out before b, then fill c with the rest of b. */
  while (j < n)
    c[k++] = b[j++];
```

```
}

void mergesort(int key[], int n)
{
  int j, k, m, *w;
  void merge(int *, int *, int *, int, int);

  /* Check to make sure that the input array is of a proper size. */
  for (m=1; m<n; m*=2)
    ;
  if (m!=n) {
    printf("ERROR: Size of the array is not a power of 2.\n");
    exit(1);
  }

  /* Allocate space for result. */
  w = (int *)calloc(n, sizeof(int));

  /* Start with single blocks in the first iteration.
   * For each iteration thereafter, increase merge size by a factor of 2.
   * So, in first interation, single elements are merged into sorted blocks
   * of 2 elements. Then, those 2 elements block are sorted into 4 element
   * blocks and so forth.
   */
  for (k=1; k<n; k*=2) {
    /* Merge subarrays partitioned according to size of k. */
    for (j=0; j<n-k; j+=2*k)
      merge(key+j, key+j+k, w+j, k, k);
    /* Copy intermediate result back into original array for next iteration. */
    for (j=0; j<n; ++j)
      key[j] = w[j];
  }
   /* Free up allocated space for intermediate array. */
  free(w);
}

#endif

driver(int size, int width, int scan)
{
  int data, addr, rpts;
  int m, i, j;
  int pre_load = -1;
  int sortlist[size-1];
  u_long* base;
  tval_t begin_time, end_time;
```

```
  /* Open the hardware interface. */
  #ifdef HARDWARE
  interfaceOpen(&base);
  #endif

 #ifdef SOFTWARE
  get_time(begin_time);
  for (rpts = 0; rpts < TIMER_RPTS; rpts++) {
 #endif

  /* Check data size. */
  for (m=1; m<size; m*=2)
    pre_load += 1;
  if (m!=size) {
    printf("ERROR: Size of array is not a power of 2.\n");
    exit(2);
  }

  /* Initialize data array, */

  for (i=0; i<size; i++) {
 #ifndef HARDWARE
    sortlist[i] = i;
 #endif

 #ifdef SIMULATION
    if (scan) addr = SCANID;
    else addr = i;
    vectorWrite(addr, i);
 #endif

 #ifdef HARDWARE
    if (scan) addr = SCANID;
    else addr = i;
    interfaceWrite(base+addr, i);
 #endif
  }

  /* Execute the sort. */

 #ifndef HARDWARE
  mergesort(sortlist, size);
 #endif

 #ifdef SOFTWARE
```

```
  }
  get_time(end_time);
#endif
  /* Reading from the top node */
  addr = size+1;

#ifdef SIMULATION
  data = 0xFFFFffff;
  for (j=0; j<pre_load; j++)
    vectorRead(addr, data);

  for (j=0; j<size; j++) {
    data = sortlist[j];
    vectorRead(addr, data);
  }
#endif

#ifdef HARDWARE
  for (j=0; j<pre_load; j++)
    interfaceRead(base+addr, data);

  for (j=0; j<size; j++) {
    interfaceRead(base+addr, data);
    sortlist[j] = data;
  }
#endif

#ifdef SOFTWARE
  print_array(sortlist, size);
#endif

#ifdef SOFTWARE
  printf("Average time per run: %g seconds\n",
 diff_time(end_time, begin_time) / ((float)TIMER_RPTS));
#endif

#ifdef HARDWARE
  interfaceClose();
#endif
}

main(int argc, char** argv)
{
  int size, width, scan;

  #ifdef SOFTWARE
```

```
char* mode = "software";
#endif

#ifdef SIMULATION
char* mode = "simulation";
#endif

#ifdef HARDWARE
char* mode = "hardware";
#endif

if (argc != 4) {
  printf("Driver program for the Merge Sort Benchmark.\n");
  printf("\n");
  printf("usage: %s <size> <width> <scan> \n", mode);
  printf("\n");
  printf("size is the number of elements to sort\n");
  printf("width is data width of elements (4 to 32)\n");
  printf("scan is either SCAN or BUS\n");
  printf("\n");
  return (-1);
};


size = atoi(argv[1]);
width = atoi(argv[2]);
scan=strcmp(argv[3], "SCAN") == 0; /* 0 for BUS, 1 for SCAN */

driver(size, width, scan);

exit(0);
}
```

A.2 library.v

```
`include "main_define.v"

`define GlobalDataHigh 32'hffffFFFF

module Merge_Top_Node (Clk, Reset, RD, WR, Addr, DataIn, DataOut,
      ScanIn, ScanOut, ScanEnable, ScanId,
      Id, In1, In2, Read1, Read2, Out);

  parameter
    DWIDTH = 32, IDWIDTH = 15, SCAN = 1;

  input Clk, Reset, RD, WR;
```

```
  input [`GlobalAddrWidth-1:0] Addr;
  input [`GlobalDataWidth-1:0] DataIn;
  output [`GlobalDataWidth-1:0] DataOut;

  input [DWIDTH-1:0] ScanIn;
  output [DWIDTH-1:0] ScanOut;
  input ScanEnable;
  input [IDWIDTH-1:0] ScanId;

  input [IDWIDTH-1:0] Id;
  input [DWIDTH-1:0] In1;
  input [DWIDTH-1:0] In2;
  output Read1, Read2;
  output [DWIDTH-1:0] Out;
  reg [DWIDTH-1:0] Out;

  assign DataOut[`GlobalDataWidth-1:0] = Out;
  assign Read1 = (RD && (Addr[IDWIDTH-1:0]==Id) && ((In1 > In2) || &In1));
  assign Read2 = (RD && (Addr[IDWIDTH-1:0]==Id) && ((In1 <= In2) || &In2));

  assign ScanEnable=(SCAN && WR && Addr[`GlobalAddrWidth-1:0]==ScanId);
  assign ScanOut= WR ? DataIn[DWIDTH-1:0]: 0;

  always @(posedge Clk)
    begin
 if (Reset)
   Out = `GlobalDataHigh;
 else if ((&In1) && (&In2))
   Out = `GlobalDataHigh;
 else if (RD && (Addr[IDWIDTH-1:0]==Id))
   if (In1 > In2)
     Out = In1;
   else
     Out = In2;
    end // always @ (posedge Clk)
endmodule // Merge_Top_Node


module Merge_Node (Clk, Reset, RD, WR, Addr, DataIn, DataOut,
  Load, Out, In1, In2, Read1, Read2);

  parameter
    DWIDTH = 32;

  inputClk, Reset, RD, WR;
  input [`GlobalAddrWidth-1:0]Addr;
```

```
  input [`GlobalDataWidth-1:0]DataIn;
  input [`GlobalDataWidth-1:0]DataOut;

  inputLoad;
  input [DWIDTH-1:0]In1;
  input [DWIDTH-1:0]In2;
  outputRead1, Read2;
  output [DWIDTH-1:0]Out;
  reg [DWIDTH-1:0]Out;

  assign Read1 = (Load && ((In1 > In2) || &In1));
  assign Read2 = (Load && ((In1 <= In2) || &In2));

always @(posedge Clk)
  begin

 /* Reset will clear the register. */

 if (Reset)
   Out = `GlobalDataHigh;
 else if ((&In1) && (&In2))
   Out = `GlobalDataHigh;
 else if (Load)
   if (In1 > In2)
     Out = In1;
   else
     Out = In2;
    end
endmodule // Merge_Node


module Merge_Low_Node (Clk, Reset, RD, WR, Addr, DataIn, DataOut,
     ScanIn, ScanOut, ScanEnable,
     Id, Load, Out);

  parameter
    DWIDTH = 32, IDWIDTH = 15, SCAN = 1;

  inputClk, Reset, RD, WR;
  input [`GlobalAddrWidth-1:0]Addr;
  input [`GlobalDataWidth-1:0]DataIn;
  input [`GlobalDataWidth-1:0]DataOut;

  /* global scan connection */
  input [DWIDTH-1:0] ScanIn;
  output [DWIDTH-1:0] ScanOut;
```

```
  input ScanEnable;

  input [IDWIDTH-1:0] Id;
  input Load;
  output [DWIDTH-1:0] Out;
  reg [DWIDTH-1:0] Out;

  assign ScanOut = Out;

  always @(posedge Clk)
    begin
 if (Reset)
   Out = `GlobalDataHigh;
 else if (SCAN && ScanEnable)
   Out = ScanIn;
 else if (!SCAN && WR && (Addr[IDWIDTH-1:0]==Id))
   Out = DataIn[DWIDTH-1:0];
 else if (Load)
   Out = 0;

    end // always @ (posedge Clk)
endmodule // Merge_Low_Node
```

A.3 generate.c

```
#include <math.h>
#include <generate_verilog.h>

#define MAXLEN 1024
#define SCANID 2000

int int_pow(int x, int y)
{
  int i;
  int ret_val = 1;
  for (i=1; i<=y; i++) {
    ret_val *= x;
  }
  return(ret_val);
}

void generate (int size, int width, int scan)
{
  int j, k, idwidth;
  char name[MAXLEN];
  char iname[MAXLEN];
  char parameters[MAXLEN];
```

```
  char connections[MAXLEN];
  char s[MAXLEN];

#ifdef OLD_SCHOOL
  char parameters[MAXLEN];
#else
  vtemplate_t template;
#endif

  idwidth = 15;

  generateHeader();

  /* Wires. */
  for (k=1; k<size; k*=2) {
    for (j=0; j<=size-k; j+=k)
      {
sprintf(name, "Level%dOut%d", k, j);
generateWire(name, width-1, 0);
sprintf(name, "Level%dLoad%d", k, j);
generateWire(name, 0, 0);
      }
  }

  /* Generate scan related wires */
  sprintf(name, "ScanEnable");
  generateWire(name, 0, 0);

  for (k=0; k<=size; k++) {
    sprintf(name, "ScanLink%d", k);
    generateWire(name, width-1, 0);
  }

  /* Low level modules. */
#ifdef OLD_SCHOOL
  sprintf(name, "Merge_Low_Node");
  sprintf(parameters, "%d, %d, %d", width, idwidth, scan);
#else
  template = declareTemplate("Merge_Low_Node", 3, "DWIDTH", "IDWIDTH", "SCAN");
#endif


  for (j=0; j<=size-1; j+=1) {
    sprintf(iname, "MLN%d", j);

    sprintf(connections, ".Id(%d'd%d), ", idwidth, j);
```

```
    sprintf(s, ".Load(Level1Load%d), ", j);
    strcat(connections, s);

    sprintf(s, ".Out(Level1Out%d), ", j);
    strcat(connections, s);

    sprintf(s, ".ScanIn(ScanLink%d), ", j);
    strcat(connections, s);

    sprintf(s, ".ScanOut(ScanLink%d), ", j+1);
    strcat(connections, s);

    sprintf(s, ".ScanEnable(ScanEnable)");
    strcat(connections, s);

#ifdef OLD_SCHOOL
    generateInstance (name, parameters, iname, connections);
#else
    generateTemplateInstance(template, iname, connections,
     width, idwidth, scan);
#endif

  }

#ifdef OLD_SCHOOL
  sprintf(name, "Merge_Node");
  sprintf(parameters, "%d", width);
#else
  template = declareTemplate("Merge_Node", 1, "DWIDTH");
#endif


  for (k=2; k<size; k*=2) {
    for (j=0; j<=size-k; j+=k)
      {
sprintf(iname, "MN%d_%d", k, j);

sprintf(connections, ".Load(Level%dLoad%d), ", k, j);

sprintf(s, ".Out(Level%dOut%d), ", k, j);
strcat(connections, s);

sprintf(s, ".In1(Level%dOut%d), ", k/2, j);
strcat(connections, s);
```

```
sprintf(s, ".In2(Level%dOut%d), ", k/2, (j+(k/2)));
strcat(connections, s);

sprintf(s, ".Read1(Level%dLoad%d), ", k/2, j);
strcat(connections, s);

sprintf(s, ".Read2(Level%dLoad%d)", k/2, (j+(k/2)));
strcat(connections, s);

#ifdef OLD_SCHOOL
generateInstance (name, parameters, iname, connections);
#else
generateTemplateInstance(template, iname, connections, width);
#endif

    }
  }

#ifdef OLD_SCHOOL
  sprintf(name, "Merge_Top_Node");
  sprintf(parameters, "%d, %d, %d", width, idwidth, scan);
#else
  template = declareTemplate("Merge_Top_Node", 3,
    "DWIDTH", "IDWIDTH", "SCAN");
#endif

  k=size/2, j=0;

  sprintf(iname, "MTN");

  sprintf(connections, ".Id(%d'd%d), ", idwidth, size+1);

  sprintf(s, ".In1(Level%dOut%d), ", k, j);
  strcat(connections, s);

  sprintf(s, ".In2(Level%dOut%d), ", k, (j+size/2));
  strcat(connections, s);

  sprintf(s, ".Read1(Level%dLoad%d), ", k, j);
  strcat(connections, s);

  sprintf(s, ".Read2(Level%dLoad%d), ", k, (j+size/2));
  strcat(connections, s);

  sprintf(s, ".ScanIn(ScanLink%d), ", size);
  strcat(connections, s);
```

```
  sprintf(s, ".ScanOut(ScanLink%d), ", 0);
  strcat(connections, s);

  sprintf(s, ".ScanId(%d'd%d), ", idwidth, SCANID);
  strcat(connections, s);

  sprintf(s, ".ScanEnable(ScanEnable)");
  strcat(connections, s);

#ifdef OLD_SCHOOL
  generateInstance (name, parameters, iname, connections);
#else
  generateTemplateInstance(template, iname, connections,
   width, idwidth, scan);
#endif

  generateTrailer();
}

main(int argc, char** argv)
{
  int size, width, scan;

  if (argc != 4) {
    printf("Computation structure generator for Mergesort benchmark.\n");
    printf("\n");
    printf("Usage: generate <size of input array> <size of data> <scan>\n");
    printf("\n");
    exit(-1);
  };

  size = atoi(argv[1]);
  width = atoi(argv[2]);
  scan=strcmp(argv[3], "SCAN") == 0; /* 0 for BUS, 1 for SCAN */

  generate(size, width, scan);

  exit(0);
}
```

Appendix B

TSP Source

B.1 driver.c

```
/* The simulated annealing solution for the TSP. */
/* Maugis Lionel contributed to this software. */


#include <stdio.h>            /* printf(), fopen() */
#include <math.h>              /* sqrt()         */
#include <string.h>            /* strdup()        */
#include <tval.h>

#define INITIAL_T                 100
#define FINAL_T                   0.1
#define COOL_RATE                    0.9 /* To lower down T (< 1). */
#define TRIES_PER_T                  500*n
#define IMPROVED_PATH_PER_T          60*n

/* Random number generator. */

#define RANDOM()      (*rand_fptr >= 0 ? *rand_fptr-- : flipCycle ())
#define two_to_the_31   ((unsigned long)0x80000000)
#define RREAL         ((double)RANDOM()/(double)two_to_the_31)

static long A[56]= {-1};
long *rand_fptr = A;

#define mod_diff(x,y)   (((x)-(y))&0x7fffffff)
long flipCycle()
{
  register long *ii,*jj;
  for (ii = &A[1], jj = &A[32]; jj <= &A[55]; ii++, jj++)
    *ii= mod_diff (*ii, *jj);

  for (jj = &A[1]; ii <= &A[55]; ii++, jj++)
    *ii= mod_diff (*ii, *jj);
  rand_fptr = &A[54];
  return A[55];
}

void initRand (long seed)
{
```

```
  register long i;
  register long prev = seed, next = 1;
  seed = prev = mod_diff (prev,0);
  A[55] = prev;
  for (i = 21; i; i = (i+21)%55)
    {
      A[i] = next;
      next = mod_diff (prev, next);
      if (seed&1) seed = 0x40000000 + (seed >> 1);
      else seed >>= 1;
      next = mod_diff (next,seed);
      prev = A[i];
    }

  for (i = 0; i < 7; i++) flipCycle();
}

long unifRand (long m)
{
  register unsigned long t = two_to_the_31 - (two_to_the_31%m);
  register long r;
  do {
    r = RANDOM();
  } while (t <= (unsigned long)r);
  return r%m;
}

/*
 * Defs
 */
#define MOD(i,n)    ((i) % (n) >= 0 ? (i) % (n) : (i) % (n) + (n))

typedef int Path[2];      /* Specify how to change path. */

/* A city is of type Point with name and position. */
typedef struct {
  float x, y;
  char *name;
} Point;

/*
 * State vars.
 */
int    n;
int    verbose = 0;
int    print = 0;
```

```
/* Contains data on all of the cities. */
Point   *cities;

/* Matrix of inter city distances. */
int     *dist;

/* iorder is the initial ordering of the cities. */
int     *iorder, *jorder;

/* Holds the x and y coordinates of a city pair. */
float   b[4];

#define D(x,y) dist[(x)*n+y]
#define SCALEX(x) (50+500.0/(b[1]-b[0])*(x - b[0]))
#define SCALEY(y) (50+700.0/(b[3]-b[2])*(y - b[2]))

float  stodeg(char *deg)
{
  int i,j,k,l,m,n,o;
  float x = 0;
  i = deg[0]=='N'||deg[0]=='E';
  j = deg[0]=='S'||deg[0]=='W';
  if (i||j) {
    ++deg;
    o = sscanf(deg, "%2d%2d%2d%2d", &k, &l, &m, &n);
    x = k * 100.0 + l * 1.0 + m /100.0 + n / 10000.0 ;
    if (j) x=-x;
  } else x = atof(deg);
  return x;
}

readCities(FILE *f)
{
  int i, j;
  int dx, dy;
  char sbuf[512];
  char sx[512], sy[512];

  /* Verify data format. */
  if (f && (fscanf(f,"%d", &n) != 1)) {
    fprintf(stderr, "Input syntax error!\n");
    exit (-1);
  }

  /* Print number of cities. */
```

```
  if (verbose) printf ("\nCities: %d\n\n", n);

  /* Initialize data structures. */
  if (!(cities = (Point*) malloc (n * sizeof(Point))) ||
      !(dist   = (int*) malloc (n * n * sizeof(int))) ||
      !(iorder = (int*) malloc (n * sizeof(int)))   ||
      !(jorder = (int*) malloc (n * sizeof(int))))
    {
      fprintf (stderr, "Memory allocation pb...\n");
      exit(-1);
    }

  /* Read in data and fill data structures. */
  for (i = 0; i < n; i++)
    {
      if (fscanf(f,"%s %s %[^\n]*", sx, sy, sbuf) != 3) {
fprintf(stderr, "Input syntax error!\n");
exit (-1);
      }
      cities[i].name = strdup(sbuf);
      cities[i].x  = stodeg(sx);
      cities[i].y  = stodeg(sy);

#define min(a,b) (a)<(b)?(a):(b)
#define max(a,b) (a)>(b)?(a):(b)
#define sqr(x)   ((x)*(x))
      if (i==0)
{
  b[0]= cities[i].x; b[1]= b[0];
  b[2]= cities[i].y; b[3]= b[2];
} else {
  b[0] = min(b[0],cities[i].x); b[1] = max(b[1],cities[i].x);
  b[2] = min(b[2],cities[i].y); b[3] = max(b[3],cities[i].y);
}
      if (verbose)
{
  printf("%s: %d %d\n", cities[i].name,
  (int)cities[i].x, (int)cities[i].y);
}
    }
  if (verbose)
    {
      printf("\n[%d %d %d %d]\n\n",(int)b[0],
      (int)b[1], (int)b[2], (int)b[3]);
    }
```

```
 /* Compute inter cities distance matrix. */
 for (i = 0; i < n; i++)
   {
     for (j = 0; j < n; j++)
{
 dx = (int)SCALEX(cities[i].x)-(int)SCALEX(cities[j].x);
 dy = (int)SCALEY(cities[i].y)-(int)SCALEY(cities[j].y);

 /* D(i,j) satisfies triangle inequality. */
 D(i,j) = sqrt ((int)(dx*dx + dy*dy));
}
   }
}

printSol()
{
 double x, y;
 int i;

 if(verbose)
   {
     for (i = 0; i <= n; i++)
{
 printf("%s  ", cities[iorder[i%n]].name);
}
     printf("\n");
   }
}

printPS(FILE *f)
{
double x, y;
int i;
fprintf (f, "%%!\n%%%%Path Length %d\n", pathLength(iorder));
fprintf (f, ".01 setlinewidth/l{lineto currentpoint 1.5 0 360 arc}def/m{moveto}def\n");
fprintf (f, "/n{currentpoint 3 -1 roll dup stringwidth pop 2 div neg 5 rmoveto show moveto}def\n");
      fprintf (f, "/s{currentpoint stroke moveto}def/Helvetica findfont\n");
      fprintf (f, "8 scalefont setfont/st{gsave show grestore}def\n");
for (i = 0; i <= n; i++)
{
 x = SCALEX(cities[iorder[i%n]].x);
 y = SCALEY(cities[iorder[i%n]].y);
 if (!i) fprintf (f, "%f %f m",x,y);
 fprintf (f, "%s (%s) %f %f l %s\n", i%20==0&i?" s\n":"",
 cities[iorder[i%n]].name,x,y,cities[iorder[i%n]].name?"n":"");
}
```

```
fprintf (f, "s showpage\n");
fflush (stdout);
}

findEulerianPath()
{
  int *mst, *arc;
  int d, i, j, k, l, a;
  float maxd;

  /* Allocate space for mst and arc. */
  if (!(mst = (int*) malloc(n * sizeof(int))) ||
      !(arc = (int*) malloc(n * sizeof(int))))
    {
      fprintf (stderr, "Malloc failed.\n");
      exit(-3);
    }

  /* Re-use vars. */
#define dis jorder

  /* Initialize variables. */
  maxd = sqr(b[1]-b[0])+sqr(b[3]-b[2]);
  d    = maxd;
  dis[0] = -1;

  for (i = 1; i < n; i++)
    {
      dis[i] = D(i,0); arc[i] = 0;
      if (d > dis[i])
{
  d = dis[i];
  j = i;
}
    }

  /*
   * O(n^2) Minimum Spanning Trees by Prim and Jarnick
   * for graphs with adjacency matrix.
   */
  for (a = 0; a < n - 1; a++)
    {
      mst[a] = j * n + arc[j]; /* Join fragment j with MST. */
      dis[j] = -1;
      d = maxd;
      for (i = 0; i < n; i++)
```

```
{
  if (dis[i] >= 0) /* Not connected yet. */
    {
      if (dis[i] > D(i,j))
{
  dis[i] = D(i,j);
  arc[i] = j;
}
      if (d > dis[i])
{
  d = dis[i];
  k = i;
}
    }
}
      j = k;
    }

  /*
   * Preorder Tour of MST.
   */
#define VISITED(x) jorder[x]
#define NQ(x) arc[l++] = x
#define DQ()  arc[--l]
#define EMPTY (l==0)

  for (i = 0; i < n; i++) VISITED(i) = 0;
  k = 0; l = 0; d = 0; NQ(0);
  while (!EMPTY)
    {
      i = DQ();
      if (!VISITED(i))
{
  iorder[k++] = i;
  VISITED(i)  = 1;
  for (j = 0; j < n - 1; j++) /* Push all kids of i. */
    {
      if (i == mst[j]%n) NQ(mst[j]/n);
    }
}
    }
  if (verbose)
    {
      for (i = 0; i < n - 1; i++)
{
  printf ("%s: %f %f ",
```

```
 cities[mst[i]/n].name,
 SCALEX(cities[mst[i]/n].x),
 SCALEY(cities[mst[i]/n].y));
 printf ("%f %f\n",
 SCALEX(cities[mst[i]%n].x),
 SCALEY(cities[mst[i]%n].y));
}

    printf("\n");
  }
 free (mst); free (arc);
}

int pathLength (int *iorder)
{
 int i, j, k;
 int len = 0;
 for (i = 0; i < n-1; i++)
  {
    len += D(iorder[i], iorder[i+1]);
  }
 len += D(iorder[n-1], iorder[0]); /* Close path. */
 return (len);
}

int getReverseCost (Path p)
{
 int a, b, c, d, e, f;

 a = iorder[MOD(p[0]-1,n)]; b = iorder[p[0]];
 c = iorder[MOD(p[0]+1,n)];

 d = iorder[MOD(p[1]-1,n)]; e = iorder[p[1]];
 f = iorder[MOD(p[1]+1,n)];

 return ( D(a,e) + D(e,c) + D(d,b) + D(b,f) -
  D(a,b) - D(b,c) - D(d,e) - D(e,f) );
 /* Add cost between c and b if non symetric TSP. */
}

doReverse(Path p)
{
 int tmp;

 tmp  = iorder[p[0]];
 iorder[p[0]] = iorder[p[1]];
```

```
  iorder[p[1]] = tmp;
}

annealing()
{
  Path   p;
  int    j, pathchg;
  int    numOnPath, numNotOnPath;
  int    pathlen, bestlen;
  double energyChange, T;

  pathlen = pathLength (iorder);
  bestlen = pathlen;

  for (T = INITIAL_T; T > FINAL_T; T = T*COOL_RATE )  /* Annealing schedule. */
    {
      pathchg = 0;
      /* For every T increment, there are several tries attempted at that T. */
      for (j = 0; j < TRIES_PER_T; j++)
{

  /* Randomly choose the cities to swap. */
  do {
    p[0] = unifRand (n);
    p[1] = unifRand (n);

    /* If both happen to be the same city, make p[1] the city after p[0]. */
    if (p[0] == p[1]) p[1] = MOD(p[0]+3,n); /* Non-empty path. */

    numOnPath = MOD(p[1]-p[0],n) + 1;
    numNotOnPath = n - numOnPath;

  } while (numOnPath < 3 || numNotOnPath < 2);

  /* Calculate energy change and decide acceptance. */
  energyChange = getReverseCost(p);

  /* Accept if the energy change is negative or with a certain
   * probability if it is not.
   */
  if (energyChange < 0 || RREAL < exp(-energyChange/T))
    {
      pathchg++;
      pathlen += energyChange;
      /* Commit the path reverse. */
      doReverse(p);
```

```
    }

  /* If the new path is shorter, that becomes the best path. */
  if (pathlen < bestlen) bestlen = pathlen;

  /* During each T, there is a max # of committed path changes.
   * If that max is exceeded, break.
   * IMPROVED_PATH_PER_T < TRIES_PER_T.
   */
  if (pathchg > IMPROVED_PATH_PER_T) break; /* Finish early. */
}

      if (verbose) printf ("T:%f L:%d B:%d C:%d\n", T, pathlen, bestlen, pathchg);
      if (pathchg == 0) break;   /* If no change, then quit. */
    }
  if (verbose) printf("\n");
}

main (int argc, char **argv)
{
  int   i;
  float r;
  FILE *f = stdin;
  FILE *k = stdin;
  FILE *l = stdin;
  FILE *m = stdin;
  long  seed = -314159L;
  tval_t begin_time, end_time;

  if (strcmp(argv[1], "-v")==0)
    verbose = 1;
  else if (strcmp(argv[1], "-p")==0)
    print = 1;
  else if (strcmp(argv[1], "-b")==0)
    {
      verbose = 1;
      print = 1;
    }
  else
    {
      fprintf(stderr,
      "Usage: %s [-V,P,B] [seed] [infile] [out] [out] [out]\n",argv[0]);
      return-2;
    }

  seed=atoi(argv[2]);
```

```
 if ((f=fopen(argv[3],"r"))==NULL)
   {
     fprintf(stderr,
     "Usage: %s [-v,p,b] [seed] [INFILE] [out] [out] [out]\n",argv[0]);
     return-3;
   }

 if (print)
   {
                        if   (   ((k=fopen(argv[4],"w"))==NULL)   ||   ((l=fopen(argv[5],"w"))==NULL)   ||
((m=fopen(argv[6],"w"))==NULL) )
{
 fprintf(stderr,
 "Usage: %s [-v,p,b] [seed] [infile] [OUT] [OUT] [OUT]\n",argv[0]);
 return-4;
}
   }

 /* Random number generator. */
 initRand (seed);

 /* Read data. */
 readCities (f);

 /* Identity permutation.
  * Traverse the list of cities in order of appearance and calculate
  * the distance travelled.
  * iorder holds the initial path.
  */
 for (i = 0; i < n; i++) iorder[i] = i;
 printPS (k);
 if (verbose)
   printf ("\nInitial Path Length: %d\n\n", pathLength(iorder));

 /* Set up first eulerian path iorder to be improved by
  * simulated annealing.
  */
 findEulerianPath();
 printPS (l);
 printSol ();
 if (verbose)
   printf ("\nApproximated Path Length: %d\n\n", pathLength(iorder));

 /* Perform annealing. */
 get_time(begin_time);
```

```
  annealing();
  get_time(end_time);
  printPS (m);
  printSol ();
  if (verbose)
    printf ("\nBest Path Length: %d\n\n", pathLength(iorder));
  fflush (stderr);

  for (i=1,r=0.0; i<n; i++)
    {
      r+=sqrt(sqr(cities[iorder[i]].x-cities[iorder[i-1]].x)+
      sqr(cities[iorder[i]].y-cities[iorder[i-1]].y));
    }
  if (verbose)
    printf ("Best Path Length in orig coord: %f\n\n", r);
  if (verbose) printf("Time: %g seconds.\n",
 diff_time(end_time, begin_time));
}
```

B.2 library.v

```
`include "main_define.v"

module TSP_Control(Clk, Reset, RD, WR, Addr, DataIn, DataOut,
  Ret_Dist, Min_Dist, Ret_Order, Min_Order);
/* OVERVIEW:   Once time has run out, this module will compare the resultss from all
 *             of the Sim_Anneal modules, and return the shortest path to the host.
 * REQUIRES:   All Sim_Anneal modules must stop computation and present the current
 *             solution.
 * EFFECTS:    Compare all TSP_Controls, find min distance, get city ordering for that
 *             min distance, and return solution to host.
 * MODIFES:    None.
 */

  parameter
    TRIES_PER_T = 250, CITIES = 10;

  // Returning the data.
  input    Clk, Reset, RD, WR;
  input [`GlobalAddrWidth-1:0]    Addr;
  input [`GlobalDataWidth-1:0]    DataIn;
  output [`GlobalDataWidth-1:0]    DataOut;

  // SIM_ANNEAL.
  //
  // Ret_Dist is an array of bits with one bit per
  // SIM_ANNEAL module.
```

```
//
// Each SIM_ANNEAL module returns a min distance
// into Min_Dist. Min_Dist will be spliced up to find
// which module had the min distance, and the appropriate
// bit in Ret_Order will be set high.
//
// Ret_Order is an array of bits similar to Ret_Dist, but
// it is actually two bits per module. One bit tells the
// module that it is in the Ret_Order phase of computation
// while the second bit tells which module should be driving
// the Min_Order bus.
//
// Min_Order will be a bus where the SIM_ANNEAL module
// with the min distance will put its ordering. The other
// modules must 'Z the bus.

input [NUM_SIM_ANNEAL*DWIDTH-1:0] Min_Dist;
input [DWIDTH*CITIES-1:0]    Min_Order;

output [NUM_SIM_ANNEAL-1:0]    Ret_Dist;
output [2*NUM_SIM_ANNEAL-1:0]    Ret_Order;

reg [NUM_SIM_ANNEAL-1:0]    Ret_Dist;
reg [CITIES*DWIDTH-1:0]    Min_Order;

reg [DWIDTH-1:0]    T;
reg [DWIDTH-1:0]    J;

reg [DWIDTH-1:0]    Out_Dist;

integer    i, j, sim_am;

if (Addr[IDWIDTH-1:0] == Id)
  begin
if (T)
  DataOut[`GlobalDataWidth-1:0] = T;
else
  DataOut[`GlobalDataWidth-1:0] = Out_Dist;
  end // if (Addr[IDWIDTH-1:0] == Id)

always@(posedge Clk)
  begin
if (Reset)
  begin
    T = 0;
    J = 0;
```

```
      State_Counter = 2;
      for (j=0, j<2*NUM_SIM_ANNEAL; j++)
  Ret_Order[j] = 0;
      for (i=0; i<NUM_SIM_ANNEAL; i++)
  Ret_Dist[i] = 0;

    end // if (Reset)
 else if (WR && (Addr[IDWIDTH-1:0]==Id))
    T=DataIn[DWIDTH-1:0];

 else if (T)
    begin
      if (J < TRIES_PER_T*CITIES)
  J++;
      else
  begin
    J = 0;
    T = T - 1;
  end // else: !if(J < TRIES_PER_T*CITIES)
    end // else: !if(WR && (Addr[IDWIDTH-1:0]==Id))

 else
    begin
      if (State_Counter == 2)
  begin
    // Ping Ret_Dist and keep it high.
    for (i=0; i<NUM_SIM_ANNEAL; i++)
Ret_Dist[i] = 1;
    State_Counter--;
  end // if (State_Counter == 2)

      if (State_Counter == 1)
  begin
    // Collect and compare min dists and ping shortest.
    length = Min_Dist[DWIDTH-1:0];
    sim_am = 1;

    for (i=2; i<=NUM_SIM_ANNEAL; i++)
if (Min_Dist[i*DWIDTH-1:(i-1)*DWIDTH] < length)
   begin
     length = Min_Dist[i*DWIDTH-1:(i-1)*DWIDTH];
     sim_am = i;
   end // if (Min_Dist[i*DWIDTH-1:(i-1)*DWIDTH] < length)

    for (j=1; j<2*NUM_SIM_ANNEAL; j=j+2)
Ret_Order[j] = 1;
```

```
    Ret_Order[2*sim_am-2] = 1;
    State_Counter--;
  end // if (State_Counter == 1)

      if (State_Counter == 0)
  DataOut = MinOrder;

    end // if (T > FINAL_T)

  endmodule // TSP_Control

module Dist_Matrix(Clk, Reset, RD, WR, Addr, DataIn, DataOut,
   Id, Index, Dist);

/* OVERVIEW:This module will hold the inter-city distances.
 *The distances will be calculated on the host from the
 *input, and will be written to this module before computation
 *begins.
 * REQUIRES:The number of cities, and thus, the SIZE of the matrix must
 *be known.
 *There must be a distance between ALL cites. If there isn't
 *a path between two cities, that distance must be tagged as such.
 *The module must know the number of Sim_Anneal modules there will
 *be. This is required because at each clock tick, the Dist_Matrix
 *will process requests from all Sim_Anneal modules at once.
 * EFFECTS:The module will return the distance between any two cites for
 *each Sim_Anneal module.
 * MODIFIES:Once the Dist_Matrix is loaded with the appropriate data, it
 *will not change.
 */

   parameter
     DWIDTH = 16, IDWIDTH = 16, SIM_ANNEAL = 4, INDEX_WIDTH = 16, CITIES = 10;

   // Loading in the data.
   input      Clk, Reset, RD, WR;
   input [`GlobalAddrWidth-1:0]      Addr;
   input [`GlobalDataWidth-1:0]      DataIn;
   output [`GlobalDataWidth-1:0]      DataOut;

   // IDWIDTH is dependent on the number of cities, and the
   // number of Sim_Anneal modules, since they all need to have
   // a unique id, but 16 bits should be more than enough.
   // Id is one greater than the number of entries in the
   // distance matrix. The dist matrix is the first to be
   // filled by the host.
```

```
  input [IDWIDTH-1:0]      Id;

  // SIM_ANNEAL.
  //
  // The width of the index will be log(n^2), n being the number of
  // cities for each Sim_Anneal module. This value will come from the host.
  // The matrix will be implemented with an array, and the
  // index into the array will be index of source city + n * index of
  // destination city. For a default of ten cities and 4 Sim_Anneal
  // modules, the width will be 24.
  input [INDEX_WIDTH*SIM_ANNEAL-1:0] Index;
  // Dist will be based on 16 bit values. Assuming a default of
  // 4 Sim_Anneal modules, the width will be 64.
  output [DWIDTH*SIM_ANNEAL-1:0]     Dist;
  reg [DWIDTH*SIM_ANNEAL-1:0]      Dist;

  // DATA STRUCTURES.
  reg [DWIDTH*CITIES*CITIES-1:0]     Dist_Matrix;

  integer     i, j;

  always@(posedge Clk)
    begin
 if (WR && (Addr[IDWIDTH-1:0]<Id))
   begin
     // Fill in the array.
     for (i=0; i<DWIDTH; i=i+1)
  Dist_Matrix[ADDR*DWIDTH + i] = DataIn[i];
   end
 else
   begin
     // Service data requests.
     // Splice the input address data to service data
     // requests from all modules.
     for (j=0; j<SIM_ANNEAL; j=j+1)
  begin
    for (i=0; i<DWIDTH; i=i+1)
Dist[j*DWIDTH+i] = Dist_Matrix[Index[INDEX_WIDTH*(j+1)-1:INDEX_WIDTH*j]*DWIDTH + i];
  end // for (i=0; i<SIM_ANNEAL; i=i+1)
   end
    end // always@ (posedge Clk)

endmodule // Dist_Matrix
module Rand(Clk, Reset, RD, WR, Addr, DataIn, DataOut,
   Result);
```

```
parameter
  SEED = -1000, CITIES = 10, DWIDTH = 32;

input Clk, Reset, RD, WR;
input [`GlobalAddrWidth-1:0] Addr;
input [`GlobalDataWidth-1:0] DataIn;
output [`GlobalDataWidth-1:0] DataOut;

output [DWIDTH-1:0] Result;
reg [DWDITH-1:0] Result;

reg [DWIDTH-1:0] prev, next, sd;
reg [56*DWIDTH-1:0] A;
reg [DWIDTH-1:0] rand_ptr;

integer i, j, k, t, r;

always@(posedge Clk)
begin
 if (Reset)

   // initRand().
   begin
 prev = SEED;
 next = 1;
 prev = prev&0x7fffffff;
 sd = prev;
 A[56*DWIDTH-1:55*DWIDTH] = prev;

 for (i = 21; i; i = (i+21)%56)
   begin
A[i*DWDITH:((i-1)*DWIDTH)]=next;
next = (prev-next)&0x7fffffff;
if (seed&&1)
  seed = 0x40000000 + (seed >> 1);
else
  seed = seed >> 1;
next = (next-seed)&0x7fffffff;
prev = A[i*DWIDTH-1:(i-1)*DWIDTH];
   end // for (i = 21; i; i = (i+21)%55)

 for (i = 0; i < 7; i++)
   // flipCycle().
   begin
for (j = 1, k = 32; k <= 56; j++, k++)
       A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:((j-1)*DWIDTH)]-A[k*DWIDTH-1:((k-
```

```
1)*DWIDTH)])&0x7fffffff;
for (k = 1; j <= 56; j++, k++)
        A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:((j-1)*DWIDTH)]-A[k*DWIDTH-1:((k-
1)*DWIDTH)])&0x7fffffff;
rand_fptr = 55;
   end // for (i = 0; i < 7; i++)

    end // if (Reset)

  else

    // unifRand().
    begin
 t = 0x80000000 - (0x80000000%CITIES);
 if (A[rand_ptr*DWIDTH-1:(rand_ptr-1)*DWIDTH] >= 0)
   r = rand_ptr--;
 else
   begin
for (j = 1, k = 32; k <= 56; j++, k++)
        A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:(j-1)*DWIDTH]-A[k*DWIDTH-1:(k-
1)*DWIDTH])&0x7fffffff;
for (k = 1; j <= 56; j++, k++)
        A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:(j-1)*DWIDTH]-A[k*DWIDTH-1:(k-
1)*DWIDTH])&0x7fffffff;
rand_fptr = 55;
r = A[56*DWIDTH-1:55*DWIDTH];
   end // else: !if(A[rand_ptr*DWIDTH-1:(rand_ptr-1)*DWIDTH] >= 0)

 while (t <= r)
   if (A[rand_ptr*DWIDTH-1:(rand_ptr-1)*DWIDTH] >= 0)
r = rand_ptr--;
   else
begin
  for (j = 1, k = 32; k <= 56; j++, k++)
          A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:(j-1)*DWIDTH]-A[k*DWIDTH-1:(k-
1)*DWIDTH])&0x7fffffff;
  for (k = 1; j <= 56; j++, k++)
          A[j*DWIDTH-1:(j-1)*DWIDTH]   =   (A[j*DWIDTH-1:(j-1)*DWIDTH]-A[k*DWIDTH-1:(k-
1)*DWIDTH])&0x7fffffff;
  rand_fptr = 55;
  r = A[56*DWIDTH-1:55*DWIDTH];
end // else: !if(A[rand_ptr*DWIDTH-1:(rand_ptr-1)*DWIDTH] >= 0)

 Result = r%m;

    end // else: !if(Reset)
end // always@ (posedge Clk)
```

```
endmodule // Rand

module Sim_Anneal(Clk, Reset, RD, WR, Addr, DataIn, DataOut,
  Id, Dist_Request, Dist, Ret_Dist, Min_Dist, Ret_Order, Min_Order, Rand);

/* OVERVIEW:This module does the actual computation. There will be several
 *modules working in parallel, and the one that produces the shortest
 *path will returned as a solution to the TSP.
 * REQUIRES:There must be a Dist_Matrix with distances between all cites.
 *The Dist_Matrix must be able to handle data requests from all
 *Sim_Anneal modules in parallel.
 *There are some parameters that the simulated annealing procedure
 *requires which must be provided by the host.
 *The module needs to know the number of cities in order to configure
 *internal data structures.
 *          Need RANDOMIZER in verilog.
 * EFFECTS:Each Sim_Anneal module will perform the simulated annealing procedure
 *on the same city data independently of the other modules. All modules
 *will be given the same parameters, but each should return a different
 *minimum distance.
 * MODIFIES:Each module will have an internal array that contains the current
 *minimum ordering of the cities. The module will modify this data
 *as the simulated annealing procedure is executed, but this should
 *not be visible outside the module until a final answer is requested.
 */

  parameter
    DWIDTH = 32, IDWIDTH = 16, INDEX_WIDTH = 16, CITIES = 10;

  // Loading in the data.
  input Clk, Reset, RD, WR;
  input [`GlobalAddrWidth-1:0] Addr;
  input [`GlobalDataWidth-1:0] DataIn;
  output [`GlobalDataWidth-1:0] DataOut;

  // The Id counts the cities as the initial ordering is loaded into the module.
  // These modules are loaded right after the Dist_Matrix.
  input [IDWIDTH-1:0] Id;

  // DIST_MATRIX.
  //
  // At every clock tick, there will be a single Dist_Request
  // between exactly two cities.
  //
  // INDEX_WIDTH is log(n^2). It just needs to be wide
  // enough to index into the dist matrix.
```

```
output [INDEX_WIDTH-1:0] Dist_Request;
reg [INDEX_WIDTH-1:0] Dist_Request;
input [DWIDTH-1:0] Dist;

// TSP_Control.
//
// Once T has decayed, TSP_Control will ping Ret_Dist
// causing computation to stop and the Sim_Anneal
// modules to return their current min distance.
// Then, the module with the lowest min distance will
// have Ret_Order go high causing it to give TSP_Control
// the min ordering.
input Ret_Dist;
output [DWIDTH-1:0] Min_Dist;
reg [DWIDTH-1:0] Min_Dist;

input [1:0] Ret_Order;
output [DWIDTH*CITIES-1:0] Min_Order;
reg [DWIDTH*CITIES-1:0] Min_Order;

input [DWIDTH-1:0] Rand;

// DATA STRUCTURES.
//
// Holds the current ordering of cities.
reg [DWIDTH*CITIES-1:0] Order;
// Holds the current min distance.
reg [DWIDTH-1:0] Cur_Dist;
// Intermediate data structures.
reg [DWIDTH-1:0] Engery_Change;
reg [DWIDTH-1:0] Index_1;
reg [DWIDTH-1:0] Index_2;
reg [DWIDTH-1:0] City_A;
reg [DWIDTH-1:0] City_B;
reg [DWIDTH-1:0] City_C;
reg [DWIDTH-1:0] City_D;
reg [DWIDTH-1:0] City_E;
reg [DWIDTH-1:0] City_F;
reg [DWIDTH-1:0] State_Counter;

integer i;
integer On_Path, Not_On_Path;

// RANDOM.

always@(posedge Clk)
```

```
    begin
if (Reset)
  // Initialize state.
  begin
    State_Counter = 11;
  else if (Ret_Order[1])
  // TSP_Control has found the node with the shortest path.
  begin
    // You are the shortest path, so return it on Min_Order.
    if (Ret_Order[0])
 // Return your order on Min_Order.
 Min_Order = Order;
    else
 // Another node found the shortest path, so just put `Z on
 // Min_Order bus.
 for (i=0; i<DWIDTH*CITIES; i=i+1)
   Min_Order[i] = z;
  end // if (Ret_Order[1])

else if (Ret_Dist)
  // Return current dist.
  Min_Dist = Cur_Dist;

else if (WR && (ADDR[IDWIDTH-1:0]>=Id) && (ADDR[IDWIDTH-1:0]<(Id+CITIES)))
  // Fill in the array with the initial ordering of cities.
  for(i=0; i<DWDITH; i=i+1)
    Order[(ADDR-Id)*DWIDTH+i] = DataIn[i];
else if (WR && (Addr[IDWIDTH-1:0] == (Id+CITIES)))
  // What is dist of initial ordering?
  for(i=0; i<DWIDTH; i=i+1)
    Cur_Dist[i] = DataIn[i];

else if (WR)
  // Do nothing and wait because other modules are being initialized.
  for (i=0; i<DWIDTH*CITIES; i=i+1)
   Min_Order[i] = z;

else
  // Perform simulated annealing.
  //
  // Randomly choose two cities. Swap them and get change
  // in energy. The change in energy will involve 8 dist
  // accesses. There will be a counter which controls state,
  // and there will be 8 registers to hold data. Each module
  // will control itself, so that there is a random pick of
  // two cities, followed by 8 dist acceses, and concluded
```

```
   // with a commit. All of the Sim_Anneal modules act
   // independently, so it doesn't matter if they are not
   // perfectly in sync.
   //
   // If path is shorter, accept. If path is longer, accept
   // with a probability that declines over time.
   begin
     if (State_Counter == 11)
  begin
    Index_1 = Rand;
    State_Counter--;
  end // if (State_Counter == 8)
     else if (State_Counter == 10)
  begin
    Index_2 = Rand;
    if (Index_1 == Index_2)
Index_2 = (Index_1+3)%CITIES;

    numOnPath = (Index_2-Index_1)%CITIES + 1;
    numNotOnPath = CITIES - numOnPath;

    if (numOnPaht < 3 || numNotOnPath < 2)
Index_1 = (Index_1+3)%CITIES;

    State_Counter--;
  end // else: !if(State_Counter == 8)
     else if (State_Counter == 9)
  begin
    City_A = Order[(Index_1-1)*DWIDTH-1:0];
    City_B = Order[Index_1*DWIDTH-1:0];
    City_C = Order[(Index_1+1)*DWIDTH-1:0];
    City_D = Order[(Index_2-1)*DWIDTH-1:0];
    City_E = Order[Index_2*DWIDTH-1:0];
    City_F = Order[(Index_2+1)*DWIDTH-1:0];
    State_Counter--;
  end // else: !if(State_Counter == 7)
     else if (State_Counter == 8)
  begin
    Dist_Request = City_A + City_B*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 6)
     else if (State_Counter == 7)
  begin
    energyChange = Dist;
    Dist_Request = City_E + City_C*CITIES;
    State_Counter--;
```

```
  end // else: !if(State_Counter == 5)
     else if (State_Counter == 6)
  begin
    energyChange = energyChange + Dist;
    Dist_Request = City_D + City_B*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 5)
  begin
    energyChange = energyChange + Dist;
    Dist_Request = City_B + City_F*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 4)
  begin
    energyChange = energyChange + Dist;
    Dist_Request = City_A + City_B*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 3)
  begin
    energyChange = energyChange - Dist;
    Dist_Request = City_B + City_C*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 2)
  begin
    energyChange = energyChange - Dist;
    Dist_Request = City_D + City_E*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 1)
  begin
    energyChange = energyChange - Dist;
    Dist_Request = City_E + City_F*CITIES;
    State_Counter--;
  end // else: !if(State_Counter == 4)
     else if (State_Counter == 0)
  begin
    energyChange = energyChange - Dist;

    if (energyChange < 0)
begin
  Cur_Dist = Cur_Dist + energyChange;
  Order[Index_1*DWIDTH-1:0] = City_E;
  Order[Index_2*DWIDTH-1:0] = City_B;
```

```
end // if (energyChange < 0 || prob)
    State_Counter = 11;
  end // else: !if(State_Counter == -2)
    end // else: !if(WR)
      end // always@ (posedge Clk)

endmodule // Sim_Anneal
```

B.3 generate.c

```
#include <math.h>
#include <generate_verilog.h>

#define MAXLEN 1024

int int_log(int val)
{
  return (int)(ceil(log((double)val)/log((double)2)));
}

void generate (int cities, int modules)
{
  int j, k, idwidth, dwidth, index_width, seed;
  int t_init = 500, t_final = 1, tries_t = 250;
  char name[MAXLEN];
  char iname[MAXLEN];
  char parameters[MAXLEN];
  char connections[MAXLEN];
  char s[MAXLEN];

  idwidth = 16;
  dwidth = 32;
  index_width = int_log(cities*cities);

  generateHeader();

  /* Wires. */
  for (j=0; j<modules; j++)
    {
      sprintf(name, "DR_%d", j);
      generateWire(name, index_width-1, 0);
      sprintf(name, "DA_%d", j);
      generateWire(name, dwidth-1, 0);
      sprintf(name, "RD_%d", j);
      generateWire(name, 0, 0);
      sprintf(name, "MD_%d", j);
      generateWire(name, dwidth-1, 0);
```

```
    sprintf(name, "RO_%d", j);
    generateWire(name, 1, 0);
    sprintf(name, "MO");
    generateWire(name, cities*dwidth-1, 0);
    sprintf(name, "R_%d", j);
    generateWire(name, dwidth-1, 0);
  }

/* Sim_Anneal. */
sprintf(name, "Sim_Anneal");
sprintf(parameters, "%d, %d, %d, %d", dwidth, idwidth, index_width, cities);

for (j=0; j<modules; j++) {
  sprintf(iname, "SA_%d", j);

  sprintf(connections, ".Id(%d'd%d), ", idwidth, ((cities*cities)+(j*cities)));

  sprintf(s, ".Dist_Request(DR_%d), ", j);
  strcat(connections, s);

  sprintf(s, ".Dist(DA_%d), ", j);
  strcat(connections, s);

  sprintf(s, ".Ret_Dist(RD_%d[0]), ", j);
  strcat(connections, s);

  sprintf(s, ".Min_Dist(MD_%d), ", j);
  strcat(connections, s);

  sprintf(s, ".Ret_Order(RO_%d), ", j);
  strcat(connections, s);

  sprintf(s, ".Min_Order(MO), ", j);
  strcat(connections, s);

  sprintf(s, ".Rand(R_%d)", j);
  strcat(connections, s);

  generateInstance (name, parameters, iname, connections);
}

/* Rand. */
for (j=0; j<modules; j++)
  {
   seed = -500 - (-500*j);
   sprintf(name, "Rand");
```

```
    sprintf(parameters, "%d, %d, %d", seed, cities, dwidth);

    sprintf(iname, "RND_%d", j);
    sprintf(connections, ".Result(R_%d)", j);
    generateInstance (name, parameters, iname, connections);
  }

/* TSP_Control. */
sprintf(name, "TSP_Control");
sprintf(parameters, "%d, %d, %d, %d", t_init, t_final, tries_t, cities);

sprintf(iname, "TSPC");

j = 0;
sprintf(connections, ".Ret_Dist[%d](RD_%d[0]), ", j);

for (j=1; j<=modules; j++)
  {
    sprintf(s, ".Ret_Dist[%d](RD_%d[0]), ", j);
    strcat(connections, s);
  }

for (j=1; j<=modules; j++)
  {
    sprintf(s, ".Min_Dist[%d:%d](MD_%d), ", j*dwidth-1, (j-1)*dwidth, j-1);
    strcat(connections, s);
  }

for (j=1; j<=modules; j++)
  {
    sprintf(s, ".Ret_Order[%d:%d](RO_%d), ", j*2-1, j*2-2, j-1);
    strcat(connections, s);
  }

sprintf(s, ".Min_Order(MO)");
strcat(connections, s);

generateInstance (name, parameters, iname, connections);

/* Dist. */
sprintf(name, "Dist_Matrix");
sprintf(parameters, "%d, %d, %d, %d, %d", dwidth, idwidth, modules, index_width, cities);

sprintf(iname, "DISTM");

sprintf(connections, ".Id(%d'd%d), ", idwidth, cities*cities-1);
```

```
  for (j=1; j<=modules; j++)
    {
      sprintf(s, ".Index[%d:%d](DR_%d), ", j*dwidth-1, (j-1)*dwidth, j-1);
      strcat(connections, s);
    }

  for (j=1; j<modules; j++)
    {
      sprintf(s, ".Dist[%d:%d](DA_%d), ", j*dwidth-1, (j-1)*dwidth, j-1);
      strcat(connections, s);
    }

  sprintf(s, ".Dist[%d:%d](DA_%d), ", j*dwidth-1, (j-1)*dwidth, j-1);

  generateInstance (name, parameters, iname, connections);

  generateTrailer();
}

main(int argc, char** argv)
{
  int cities, modules;

  if (argc != 3) {
    printf("Computation structure generator for TSP benchmark.\n");
    printf("\n");
    printf("Usage: generate <number of cities> <number of modules>\n");
    printf("\n");
    exit(-1);
  };

  cities = atoi(argv[1]);
  modules = atoi(argv[2]);

  generate(cities, modules);

  exit(0);
}
```

Appendix C

Data Sets

C.1 80 City TSP

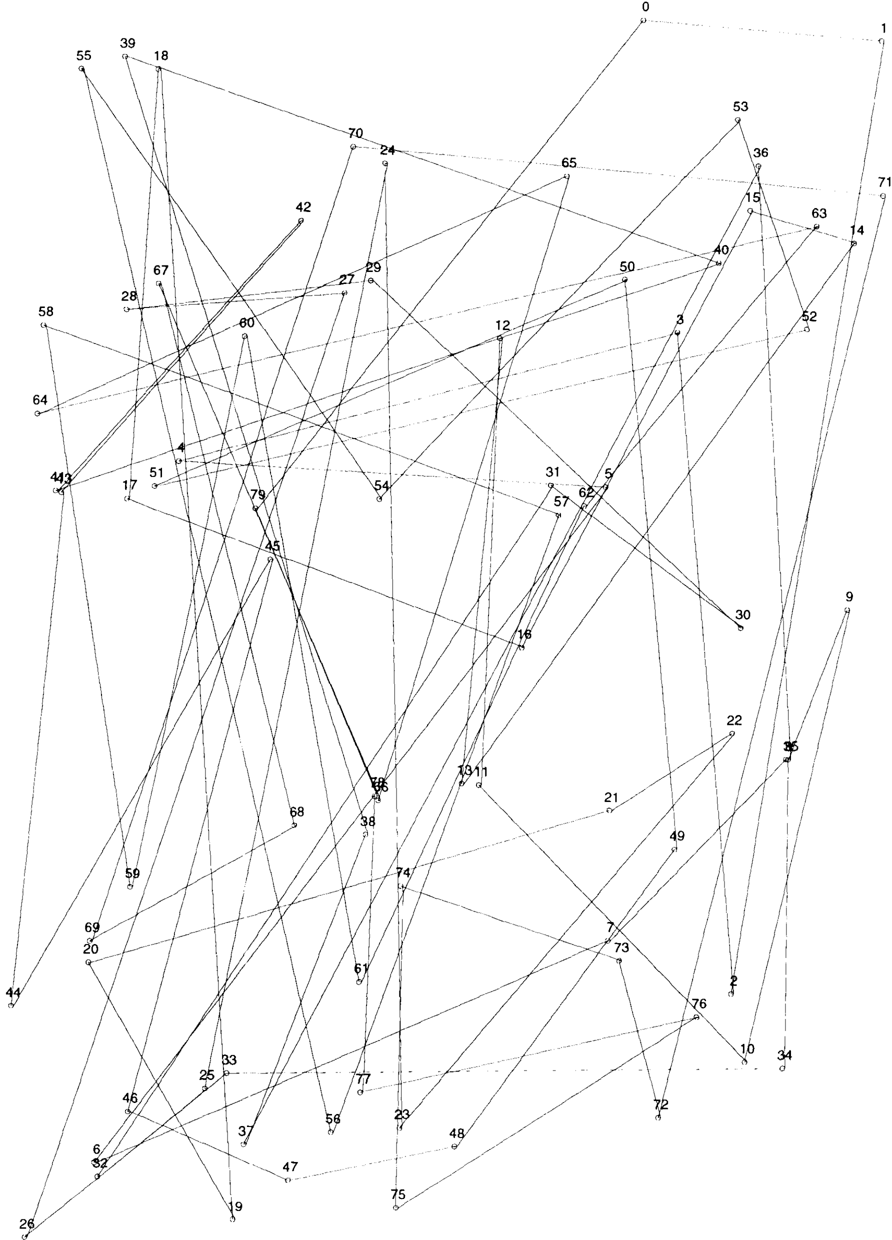


Figure C.1: 80 City TSP

C.2 640 City TSP

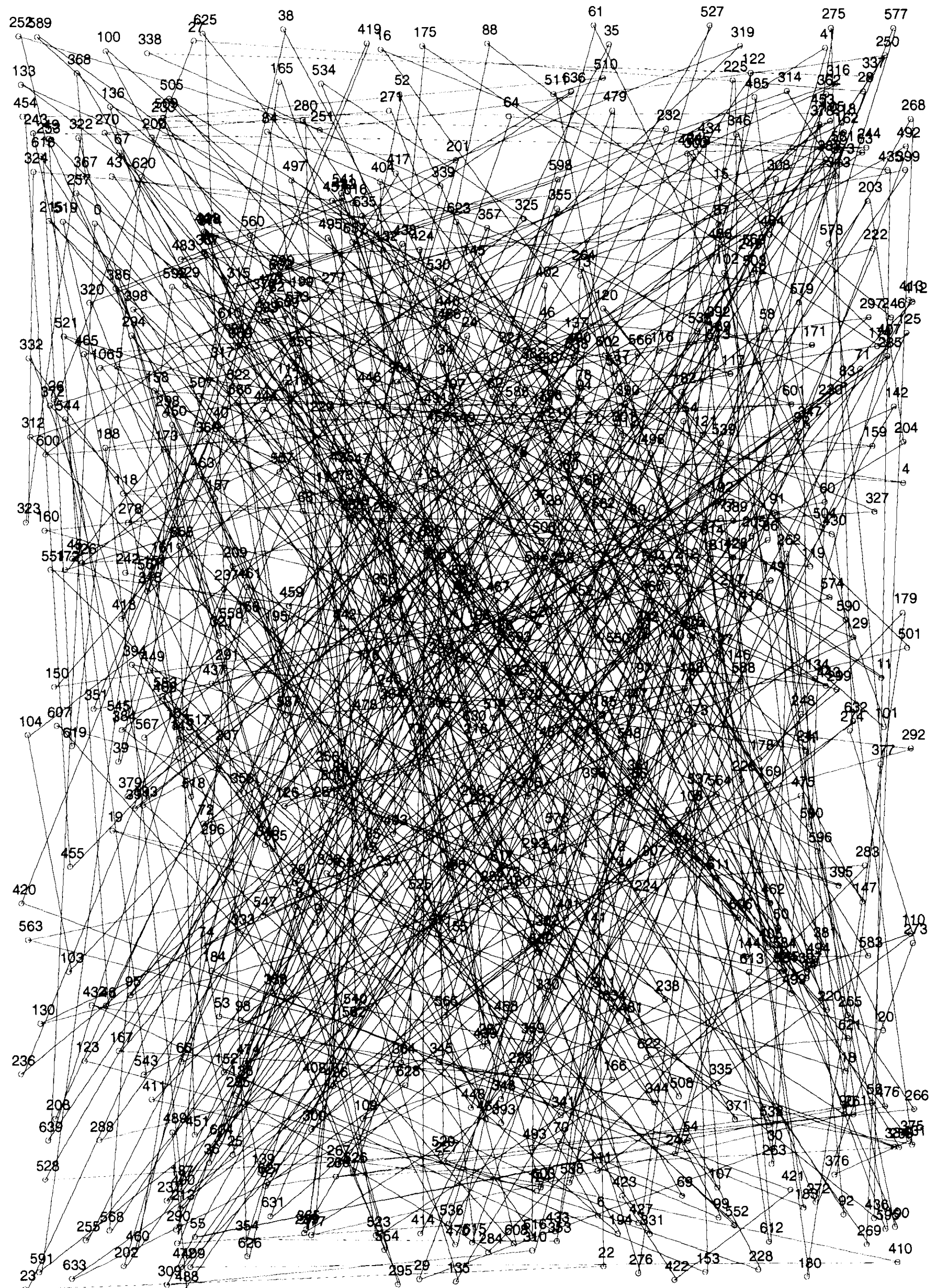


Figure C.2: 640 City TSP

Appendix D

Results of Base Case

D.1 80 City TSP Solution

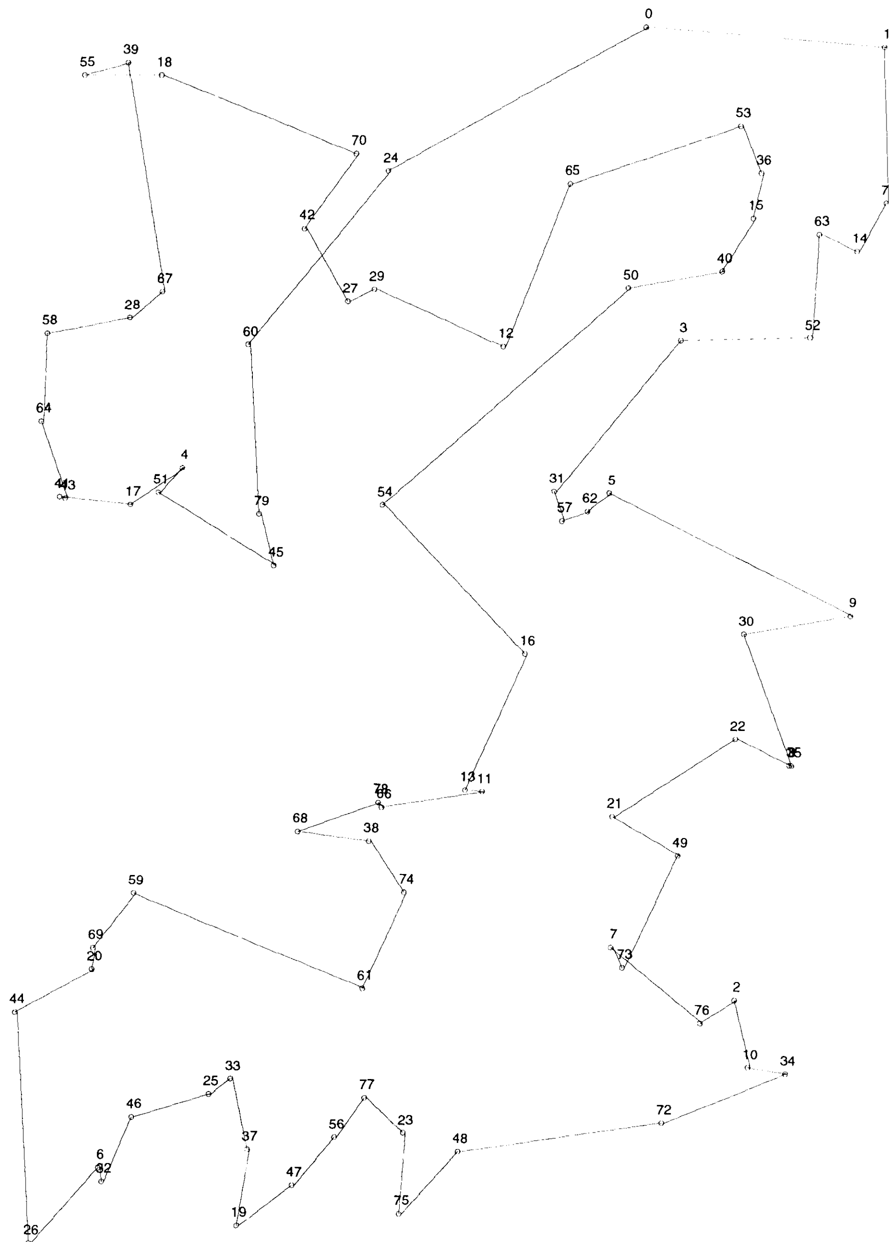


Figure D.1: Solution for 80 City TSP

D.2 640 City TSP Solution

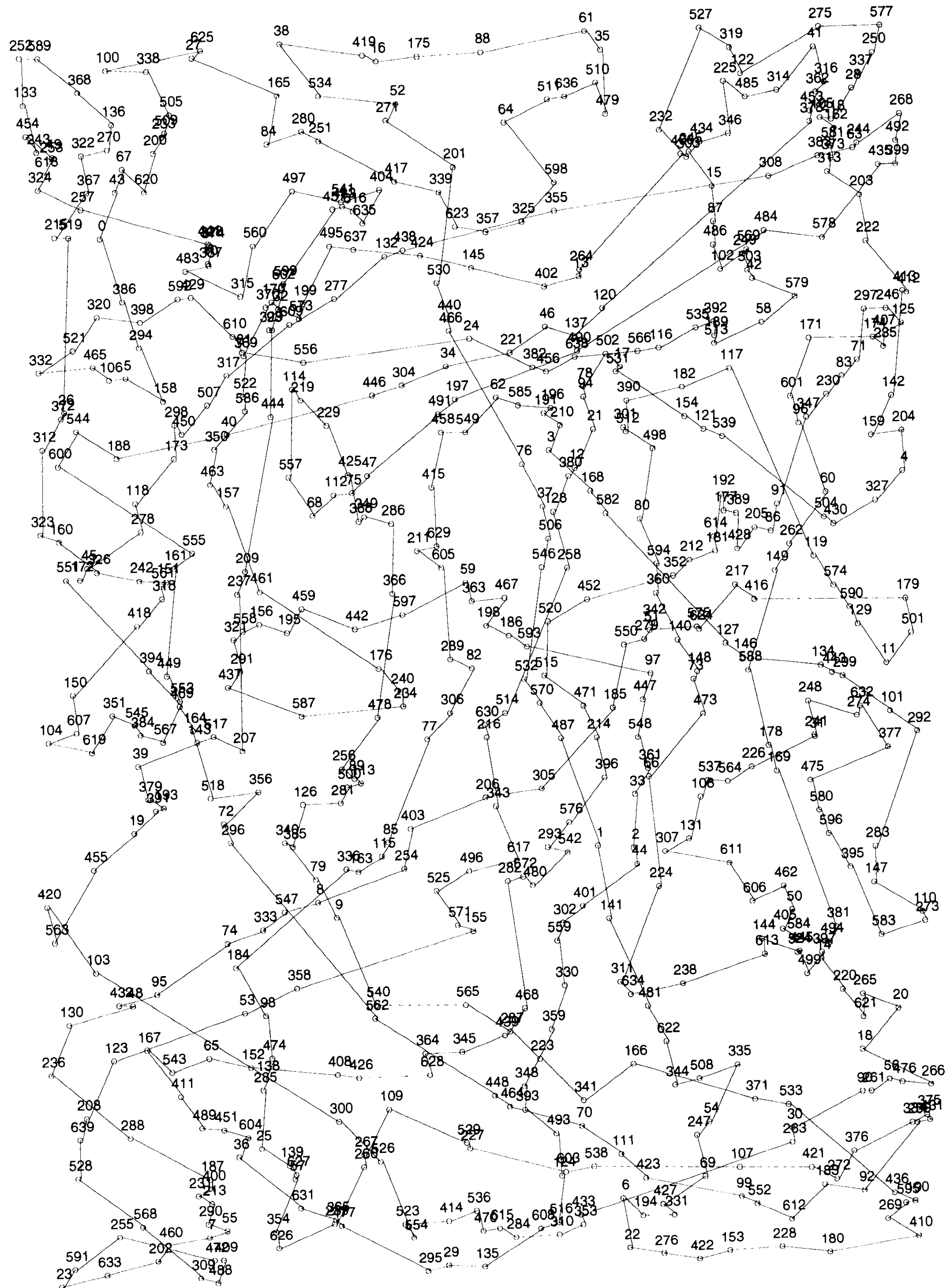


Figure D.2: Solution for 640 City TSP

References

[1] J.M. Arnold. The Splash 2 Software Environment. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 88-93, Napa, CA, April 1993. IEEE.

[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] J. Babb, M. Frank, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, P. Finch, A. Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, April 1996. IEEE.

[4] J. Babb, M. Frank. Solving Graph Problems with Dynamic Computation Structures. In *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, Boston, MA, Nov. 1996.

[5] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142-151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, Jan. 1993.

[6] R. Barua. Prospects for FPGA-based Reprogrammable Systems. MIT/LCS, Sept. 1996.

[7] P. Bertin and H. Toutati. PAM Programming Environments: Practice and Experience. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 133-138, April 1994.

[8] A. Brilliot. NAPA Architecture Version 1.0 Specification. *National Semiconductor Corporation* (1995).

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press. Cambridge, MA. 1990.

[10] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal. Emulation of a Sparc Microprocessor with the MIT Virtual Wires Emulation System. MIT/LCS, Jan. 1994.

[11] DAWN VME Products. *SLIC Evaluation Board User's Guide for DAWN VME PRODUCTS SLIC EB-1 Version 1.0*, 1993.

[12] A. DeHon. DPGA-coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 31-39, Napa, CA, April 1994. IEEE.

[13] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, 24(1), Jan. 1991.

[14] M. Gokhale, J. Kaba, A. Marks, and J. Kim. A Malleable Architecture Generator for FPGA Computing. *Proceedings SPIE Workshop on High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic,* pages 208-217, Boston, MA, November 1992. SPIE.

[15] M. Gokhale and R. Minnich. FPGA Computing in Data Parallel C. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 94-100, Napa, CA, April 1993. IEEE.

[16] M. Gokhale and J. Schlesinger. A Data Parallel C and its Platforms. *Proceedings, Frontiers'95, The Fifth Symposium on the Frontiers of Massively Parallel Computation* (1995).

[17] M. Gokhale and B. Schott. Data Parallel C on a Reconfigurable Logic Array. *Supercomputing Research Center Technical Report SRC-TR-94-121* (1994).

[18] S. Goldwasser. Approximation Algorithms. *6.045J/18.400J: Automata, Computability and Complexity Handout 40.* MIT. May 7, 1997.

[19] IKOS Systems, Inc. *VirtuaLogic Emulation System Documentation, 1996.* Version 1.2.

[20] C. Iseli and E. Sanchez. Spyder: A Reconfigurable VLIW Processor Using FPGAs. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 17-24, Napa, CA, April 1993. IEEE.

[21] J. Little, et. al. An Algorithm for the Travelling Salesman Problem.

[22] J. Mohan. Experience with Two Parallel Programs Solving the Travelling Salesman Problem. In *Proceedings of the International Conference on Parallel Processing* (ICPP), pages 191-193, 1983.

[23] G. Reinelt. The Travelling Salesman: Computational Solutions for TSP Applications. *Lecture Notes in Computer Science.* Springer-Verlag. Berlin, Germany, 1994.

[24] Synopsys, Inc. *Command Reference Manual*, Version 3.0, Dec. 1992.

[25] D. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, Boston, 1991.

[26] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal. Baring It All to Software: The RAW Machine. *MIT Laboratory for Computer Science TR-709*, March 1997.

[27] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. Prism-ii Compiler and Architecture. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 9-16, Napa, CA, April 1993. IEEE.

[28] M.J. Wirthlin, B.L. Hutchings, and K.L. Gilson. The Nano Processor: A Low Resource Reconfigurable Processor. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 23-30, Napa, CA, April 1994. IEEE.

5466.21