Storage Area Network Optimization

A cooperation of *Ancor Communications, Minneapolis, USA* and *Freie Universität Berlin, Germany.*

Final Report

HELMUT ALT
STEFAN FELSNER
LUDMILA SCHARF

Freie Universität Berlin
Institut für Informatik
Takustr. 9
14195 Berlin, Germany

felsner@inf.fu-berlin.de

November 8, 2000

Contents

1 Introduction

The Storage Area Network Optimization Project (SANO) is a cooperation of *Ancor Communications* (now *QLogic*) and *Freie Universität Berlin*.

The information explosion and the need for high-performance communications for server-to-storage and server-to-server networking have been the focus of much attention during the 90s. Performance improvements in storage, processors, and workstations, along with the move to distributed architectures such as client/server, have spawned increasingly data-intensive and high-speed networking applications. The interconnect between these systems and their input/output devices demands a new level of performance in reliability, speed, and distance. Fibre Channel, a highly-reliable, gigabit interconnect technology allows concurrent communications among workstations, mainframes, servers, data storage systems, and other peripherals using SCSI and IP protocols. It provides interconnect systems for multiple topologies that can scale to a total system bandwidth on the order of a terabit per second. Fibre Channel delivers a new level of reliability and throughput. Switches, hubs, storage systems, storage devices, and adapters are among the products that are on the market today, providing the ability to implement a total system solution.

A Storage Area Network (SAN) is a network behind the servers linking one or more servers to one or more storage systems. *QLogic* offers a broad product line of SAN infrastructure (see[1]). One of them is a SANbox – switch based on fibre channel technology (see Figure 1).

QLogic offers SANboxes in 8-, 16-, 64-, and 128-port configuration. The subject of this project were 64- and 128-type networks. The main task in the Storage Area Network Optimization Project (SANO) was the design and implementation of off-line path allocation algorithms for these multi-stage fibre channel networks.

The path allocation problem arises because routes between ports of a SANbox are static, i.e. can not be changed dynamic at real time. The limit is due to the fiber channel specification. Due to the static allocation of routes it can happen that a set of ports cause a lot of traffic on a specific connection thus forcing buffering and delay, while other connections are are underloaded and would be able to carry more traffic. It is though possible to change routes within a SANbox, but the box has to be taken offline for that task.

The basic question adressed in the project was the following.

Figure 1: possible usage of a SANbox

- Assuming a certain communication demand between ports of the box design and evaluate algorithms for path allocation that make the need for buffering unlikely.

The first phase of the project was used to realize a small pilot-project. The aim of this was to get some feeling for the problem so that we can make suggestions on how to proceed in the main phase of the project.

In the main phase routing algorithms for all in project considered network types were implemented, and integer program models for optimal solutions with Cplex were designed.

The implemented system consists of several parts which can be described as follows:

1. The design of data-structures to model inputs and routings on two of the network topologies supported by Ancor (Type 64 and Type 128).

2. The implementation of input models, i.e., assignment of a communication-demand to each pair of ports.

3. The design and implementation of routing heuristics, i.e., assignment of a communication-path to each pair of ports.

4. The implementation of an evaluation tool that allows to do some statistics and compares the objective value of solutions produced by the

heuristics with a lower bound.

5. The design and implementation of an integer program model for finding optimal solutions.

Close to the end of the project the problem became a focus shift. In discussion with engeneers from Ancor it was realized that it is very unlikely to have any good guess ore measurement of port-to-port communication demands. At best it would be feasible to measure the load at each port disregarding the corresponding sender/receiver port. In a third phase we tried some ideas to attack the new problem. In particular we analyzed ways of getting a guess of port-to-port communication from the data of the port loads. And of improving a path assignment given the data of port loads and some information about congestions. This last part of the project will only be described quite superficial in section 5 of this report.

2 Problem

2.1 Model for the problem

We consider the Type 64 and Type 128 networks which have 64 or 128 ports respectively.

For the type 64 network a problem instance (input) can be considered as a 64×64 matrix $R = (r_{i,j})$ where $r_{i,j}$ is the load imposed to the network by the communication between ports i and j, we will call $r_{i,j}$ the *demand* of pair (i, j). An input-generating algorithm will usually produce demands between zero and one, i.e., $0 \leq r_{i,j} \leq 1$.

The network consists of three layers of routing-switches, see Figure 2. Though the first and third layer are equal we still, by historical reasons, denote the switches on these layers by M and D. Each of the eight outer-switches M_s, D_u has 2×8 ports, so it can be connected to 8 machines or disks on the out side, and doubly connected to each of the 4 switches S_t from the intermediate-layer on the other side. A legal path for the demand $r_{i,j}$ can be described by specifying a pair of ports: on the outer switch and on an intermediate-switch used by the path. We denote the path assigned to the pair (i, j) by algorithm A as $P_A(i, j)$. The *load of an edge E* under assignment A and a given demand R is defined as

$$\lambda(E) = \sum \{r_{i,j} : P_A(i, j) \text{ uses } E\}.$$

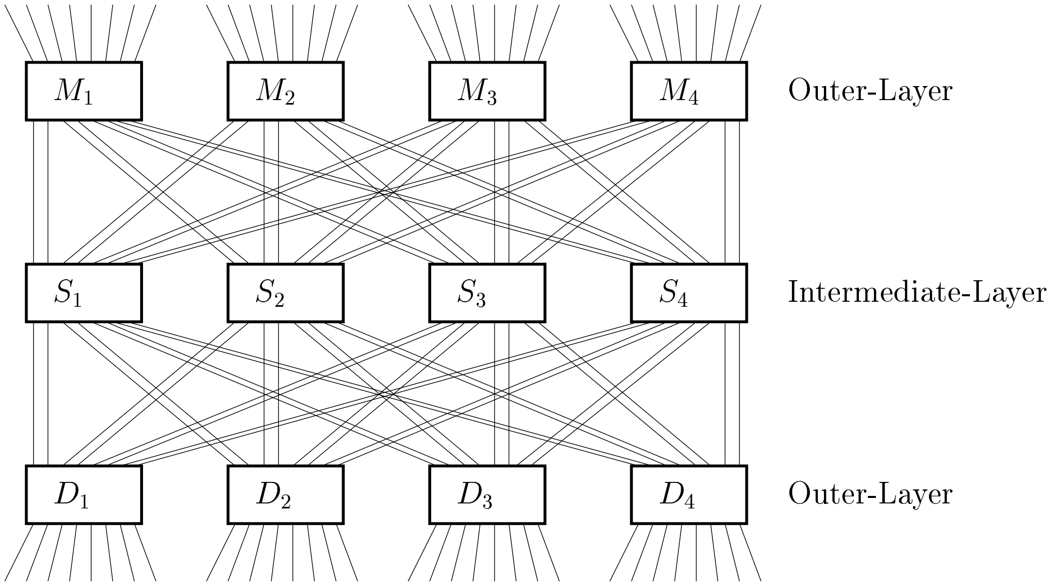

Figure 2: The Type 64 network.

The aim of our algorithms is to keep the load on all edges small. This intention is reflected by defining the *load of a path assignment* A as

$$\Lambda(A) = \max\{\lambda(E) : E \text{ edge }\}.$$

Type 128 network is build analog, with 16 outer switches and 8 intermediate switches, each of the same type as above, so it has 128 outer ports (ports), and single connections between outer and intermediate-switches. In the case of single connections the path for one demand can be described by an intermediate switch, through which the demand is routed.

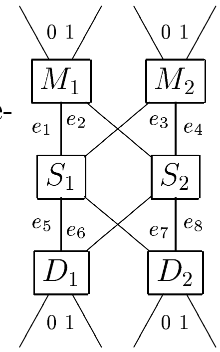

Figure 3: type 4-4 network

We illustrate the methods and algorithms we used for these complex networks on a small example network with four outer switches and two intermediate switches, each having 2×2 ports. Let call it a Type 4-4 network (Figure 3). We will furtheron explain important concepts and algorithms at the Type 4-4 network.

Further simplifying the problem we assume that the only non-zero demands are those from ports on the upper side to ports on the lower side, i.e. there is only traffic from switches M_i to switches D_j, $i, j = 1, 2$. This simplification is suggested by the idea that the workstations (machines) are connected to one side of the network (ports of M-switches) and storage devices (discs) are connected the other side (to ports of D-switches). The assumption about the demands then corresponds to the assumption that there is no workstation-to-workstation or disc-to-disc communication.

The input matrix R is thus reduced from 8×8 to 4×4. This small size will allow us to be very concrete and allow to clearly trace what's going on behind the scenes.

A matrix instance R:

R	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	$0,95$	$0,04$	$0,07$	$0,95$
$D_1:1$	$0,02$	$0,05$	$0,06$	$0,01$
$D_2:0$	$0,03$	$0,06$	$0,05$	$0,02$
$D_2:1$	$0,01$	$0,07$	$0,04$	$0,03$

is a problem instance for the Type 4-4 network.

A possible path assignment is described by a matrix P:

P	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	S_1	S_1	S_2	S_2
$D_1:1$	S_1	S_1	S_2	S_2
$D_2:0$	S_2	S_2	S_1	S_1
$D_2:1$	S_2	S_2	S_1	S_1

From P we read off that all traffic from M_1 to D_1 is routed through intermediate switch S_1, traffic from M_1 to D_2 is routed trough S_2, from M_2 to D_1 through S_2, and from M_2 to D_2 through S_1. This results in following edge loads:

E	$\lambda(E)$
e_1	$0,95 + 0,02 + 0,04 + 0,05 = 1.06$
e_2	$0,03 + 0,01 + 0,06 + 0,07 = 0.17$
e_3	$0,05 + 0,04 + 0,02 + 0,03 = 0.14$
e_4	$0,07 + 0,06 + 0,95 + 0,01 = 1.09$
e_5	$0,95 + 0,02 + 0,04 + 0,05 = 1.06$
e_6	$0,07 + 0,06 + 0,95 + 0,01 = 1.09$
e_7	$0,05 + 0,04 + 0,02 + 0,03 = 0.14$
e_8	$0,03 + 0,01 + 0,06 + 0,07 = 0.17$

The load of the path assignment P is the maximum of these values, i.e., 1.09.

2.2 Lower bounds and complexity of the problem

For the simplest lower bound define $\rho(R) = \sum_{(i,j)} r_{i,j}$, i.e., as the sum of all demands. Clearly this overall demand has to be brought through the network. The load on a path contributes to the load of two edges. Since there are 64 connecting edges we find that the maximum load of an edge is at least $2 \cdot \rho(R)/64 = \rho(R)/32$. In other words $\Lambda(A)\rho(R)/32$ for every assignment A. In general this bound could at best be attained if we give up on the requirement that the demand for (i,j) has to use just one path and would instead allow this demand to spread over several intermediate switches. The relaxation of the discrete nature of the demands can also be used in better lower bounds.

For our Type 4-4 network and the input matrix R introduced in section 2.1, we evaluate $\rho(R) = 2.46$. Since our network is directed there are only four edges at the upper and four at the lower level the load can be distributed over. The simple lower bound then becomes $\frac{\rho(R)}{4} = 0.615$.

To improve uppon this bound we consider the eight edges incident to each outer-switch separately, let $\rho(M_s, R) = \sum_{(i,j),i \in M_s} r_{i,j} + \sum_{(i,j),j \in M_s} r_{i,j}$. The overall bound $\rho^*(R)$ is obtained as 1/8 of the maximum of the eight values $\rho(M_s, R)$, $\rho(D_u, R)$ for $s, u = 1, .., 4$. This lower bound is the one we actually use when evaluating the quality of algorithms for the path assignment. Note that this bound can be as low as 1/8 of the load of an optimal assignment. This bad performance of the lower bound happens if for some $s \in [1..4]$ the whole load $\rho(M_s, R)$ is caused by the demand between just one pair of ports and all other demands are 0.

In our example Type 4-4-network with input R we have to calculate:

$$\begin{aligned}
\rho(M_s, R) &= \sum_{i=1,2,j=0,1,k=0,1} r_{M_s:k,D_i:j} \qquad \text{for } s = 1, 2 \\
\rho(D_s, R) &= \sum_{i=1,2,j=0,1,k=0,1} r_{M_i:j,D_s:k} \qquad \text{for } s = 1, 2.
\end{aligned}$$

The demand represented by any one of these values has to be routed over one of two edges. Therfore the maximum of the four values, divided by two is a lower bound. This lower bound can still be as bad as 1/2 of real optimum.

For our input matrix R the values are:

	ρ	max	max /2
M_1	1.23		
M_2	1.23		
D_1	2.15	2.15	1.075
D_2	0.31		

We have thus gotten a much better lower bound than before.

Further increase in the lower bound is possible by solving a packing problem for each outer-switch separately. More concretely the problem solved for a switch M_s is to assign each of the demands $r_{i,j}$, where i is a port at switch M_s, as a whole to one of the four edges incident to M_s.

If we solve this packing problem for switch D_1 we get following optimal loads on edges $e_5,\ e_6$:

$$\begin{array}{c|l} e_5 & 0.95 + 0.07 + 0.04 + 0.02 = 1.08 \\ e_6 & 0.95 + 0.06 + 0.05 + 0.01 = 1.07 \end{array}$$

In this small network one can see, D_1 has the biggest demands, the three other switches would lead to weaker bounds. Note that again we got a better lower bound than with previous method. The difference of the lower bound of 1.08 and the value 1.09 acheived by the concrete path assignment P has become quite small.

The packing problem that has to be solved for the lower bound discussed last is a hard problem. Concretely, the decission version of the optimization problem is NP-complete when the number n of opposite ports (number of j's) is part of the input. A possible reduction is from PARTITION. As a consequence we note that our routing problem is also NP-complete when considered on an appropriately growing family of networks. Nevertheless, the packing lower bound is not completely useless, since the problem that has to be solved for each switch can be approximated up to a factor of 1.75 [Graham's list scheduling].

Solving the packing problem for all switches does not solve the routing problem as whole. This is because the optimal routes for each of two switches in source-destination pair do not necessarily meet at the same intermediate switch. This is indicated with Figure **??**.

3 Solutions

3.1 Input generators

Unfortunately, we did not have a data set of real-world demands for the routing problem. To test algorithms we thus had to generate some artificial inputs. To have a good basis for a judgement of the strength and weaknesses of the algorithms we decided to produce several different sets of inputs. In particular we were interested in inputs that are hard to route, i.e., where the load of the path assignments produced by the algorithms were large compared to lower bound. We decided to perform tests with the following three input generators:

UNIFORM DISTRIBUTION: Choose each $r_{i,j}$ uniformly at random from $[0, 1]$.

1 WITH PROBABILITY p: Each $r_{i,j}$ is set to 1 with probability p, with the remaining probability $1 - p$ the demand $r_{i,j}$ is 0.

LARGE WITH PROBABILITY p AND FACTOR f: The value for $r_{i,j}$ is set in two steps.

- $r_{i,j}$ is chosen uniformly at random from $[0, 1]$
- with probability p blow up $r_{i,j}$ by a factor of f (i.e., $r_{i,j} \leftarrow f \cdot r_{i,j}$) with the remaining probability $1 - p$ leave $r_{i,j}$ unchanged.

We experimented with some other input strategies. They proved to be not harder to route than 1 WITH PROBABILITY p or LARGE WITH PROBABILITY p AND FACTOR f. Those two input strategies were selected to be "difficult" compared to lower bound.

According to our observation it is fairly easy to produce a good (almost optimal) path assignment if one of the following conditions is fulfilled.

1. demands do not vary strongly. (In this case a routing can distribute the load evenly over the network – even a random assignment will be very good in this case.)
2. a large fraction of the overall load is produced by a small number of large demands. (In this case a good routing only has has to keep the paths for few large demands as disjoint as possible.)

These considerations show that distributions considered in probability theory, e.g., normal-distribution, will not lead to interestig input data. In these cases the demands are very concentrated, i.e. most demands come from a small interval.

3.2 Heuristics

We now describe the algorithmic ideas used to generate routings for a given input set R representing demands.

RANDOM: Assign the intermediate switch for path $P(i,j)$ uniformly at random, i.e., without considering the demand $r_{i,j}$.

Though the algorithmic idea of this heuristic is everythig but deep it helps in evaluating inputs and other heuristics. More concretely, if a random routing is likely to give good path assignments on some input class, then the inputs can be considered easy.

A random path assignment for the Type 4-4 network:

$P(Rand)$	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	S_1	S_2	S_2	S_1
$D_1:1$	S_1	S_1	S_1	S_1
$D_2:0$	S_1	S_1	S_2	S_2
$D_2:1$	S_2	S_1	S_1	S_1

Evaluated with our example demands we find $\Lambda(Rand) = 2.04$, the maximum load being on edge e_5.

FIRST FIT (FF): This heuristic is guided by the a global parameter C, which models the capacity of an edge. Suppose there is a switch S_t such that after routing $P(i,j)$ through S_t the load of both edges used by $P(i,j)$ remains below C. In this case FF will use the least indexed intermediate-switch with this property. If there is no such switch FF will assign $P(i,j)$ such that the maximal load after the assignment is minimal.

Since again we had no real basis for the specification of a value for C. We decided to choose

$$C(R) = \max_{\substack{i \text{ port of } M_s \\ j \text{ port of } D_k}} \{\rho(M_s, R), \rho(D_k, R)\} \; s, k = 1, 2$$

thus depending on the input matrix R or more precisely on the demands between pairs of outer switches, we call such a set of demands a demand group.

In our example network, there are 4 demand groups:

1. $M_1 \to D_1 \Rightarrow C_1 = 1.075$
2. $M_1 \to D_2 \Rightarrow C_2 = 0.615$
3. $M_2 \to D_1 \Rightarrow C_3 = 1.075$
4. $M_2 \to D_2 \Rightarrow C_4 = 0.615$

Let us look at a run of the FirstFit heuristic. The algorithm would route the demands $M_1 : 0 \to D_1 : 0$ and $M_1 : 0 \to D_1 : 1$ over the switch S_1 with load of 0.97 on edges e_1, e_5 staying below the "capacity" $C = C_1$.

The next two demands $M_1 : 0 \to D_2 : 0$ and $M_1 : 0 \to D_2 : 1$ would be routed over switch S_2, since edge e_1 to switch S_1 is overloaded according to capacity C. This leaves edges e_2 and e_8 with load 0.04.

The next four demands: $M_1 : 1 \to D_2 : 0$ through $M_1 : 1 \to D_2 : 1$, would be routed the same way, with first two leaving edges e_1, e_5 with load 1.06 under capacity C, and the next two taking switch S_2 due to capacity C, thus causing load of 0.17 on edges e_2, e_8.

The traffic path from $M_2 : 0$ to both ports of D_1 goes through S_2, because each of two demands would overload the edge e_5. After these steps edges e_4, e_6 have load 0.13.

The next two paths from $M_2 : 0$ to ports of D_2 go over switch S_1 because the load 0.09 on involved edges e_3, e_7 stays under capacity C.

The next requirement $M_2 : 1 \to D_1 : 0$ causes overflow on both possible paths: taking switch S_1 would result in load 2.01 on edge e_5, path over S_2 would leave edges e_4, e_6 with load 1.08. The algorithm chooses the second path because it has lesser overflow.

The rest demands would be routed over S_1, because each leaves edge loads under capacities C and C respectively.

The maximal load of this path assignment is $\Lambda(FF) = 1.08$.

Best Fit (BF): This heuristic also uses the capacity C of an edge. BF gives preference to a switch S_t such that after routing $P(i,j)$ through S_t the load of both edges used by $P(i,j)$ remains below C but the load of one of these edges comes as close to C as possible. If all assignments violate the capacity then BF will assign $P(i,j)$ such that the maximal load after the assignment is minimal.

For our Type 4-4 network this heuristic would produce the following path assignment:

$P(BF)$	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	S_2	S_2	S_2	S_1
$D_1:1$	S_2	S_2	S_1	S_1
$D_2:0$	S_1	S_1	S_1	S_2
$D_2:1$	S_1	S_1	S_1	S_2

with $\Lambda(BF) = 1.13$ on edge e_6.

Best Balance (BB): BB assigns $P(i,j)$ to the path which has the least load on the heavier-loaded of the two edges of the path.

According to this heuristic the first demand $M_1 : 0 \rightarrow D_1 : 0$ can be assigned to any of two possible paths since they are equally loaded. The implementation of the algorithm chooses the last of equally loaded paths, in this case one over S_2.

The next 9 demands have either e_2 or e_6 as parts of one of two possible ways. All these demands are much smaller then the first one, thus the load on an alternative path stays under this on path with edges e_2 or e_6. So they will all be routed over S_1.

The alternatives for next two requirements $M_2 : 0 \rightarrow D_2 : 0 = 0.5$ and $M_2 : 0 \rightarrow D_2 : 1 = 0.4$ are:

- e_3, e_7 with maximal current load 0.17, and
- e_4, e_8 with no current load.

So they are both routed over the second path.

For the next demand $M_1 : 1 \rightarrow D_1 : 0$ chooses the algorithm the path e_3, e_5 with load 0.24 against the path e_4, e_6 with current load 0.95. That increases load on e_5 to 1.19. The next traffic $M_1 : 1 \rightarrow D_1 : 1$ has the same alternatives and will be routed over now less loaded way e_4, e_6.

The last two requirements take the least loaded path e_4, e_8. The path matrix is now:

$P(BB)$	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	S_2	S_1	S_1	S_1
$D_1:1$	S_1	S_1	S_1	S_2
$D_2:0$	S_1	S_1	S_2	S_2
$D_2:1$	S_1	S_1	S_2	S_2

with $\Lambda(BB) = 1.19$ on edge e_5.

SORTED BEST BALANCE (SOBB): The assignment of the route for $P(i, j)$ is guided by the same rule as in algorithm BB. The difference is that the pairs (i, j) are considered in order of decreasing demand.

Sorting of demands avoids the effect we have seen in example above, that the edges get first loaded with many small demands, and the large demands come on top of those. So we got some heavy weighted paths, and some light weighted with obvious optimization opportunities.

For our network the same algorithm as above with prior sorted demands produces following paths:

$P(SoBB)$	$M_1:0$	$M_1:1$	$M_2:0$	$M_2:1$
$D_1:0$	S_2	S_2	S_2	S_1
$D_1:1$	S_2	S_1	S_1	S_1
$D_2:0$	S_1	S_1	S_2	S_2
$D_2:1$	S_1	S_1	S_2	S_2

with $\Lambda(SoBB) = 1.08$ on edge e_6.

For this example first fit and sorted best balance heuristics produce routes with the same maximal load, which is in this case also the optimum. In most cases though the best balance algorithm with prior sorted demands produces better results.

k-OPTIMUM (k-OPT): The route for the input R is first assigned using one of the heuristics above (e.g. random). Then one of the demand on the heaviest edge will be rerouted over some other edge pair. After k reroutings are performed, the new path assignment is evaluated and if it is better, the old one is replaced. The algorithm terminates when all possible reroutings were performed and no improvement was achieved.

The simplest k-Opt is 1-opt: one of the demands gets assigned a new path. The new route is evaluated and if it got better the old one is substituted.

Let us take the random route assignment for our example network as start position. Recall that the random heuristic left us with the edge loads shown in the following table:

Edge	Load	Demand
e_1	1.18	$M_1 : 0 \to D_1 : 0$, $M_1 : 0 \to D_1 : 1$, $M_1 : 0 \to D_2 : 0$, $M_1 : 1 \to D_1 : 1$, $M_1 : 1 \to D_2 : 0$, $M_1 : 1 \to D_2 : 1$
e_2	0.05	$M_1 : 0 \to D_2 : 1$, $M_1 : 1 \to D_1 : 0$
e_3	1.09	$M_2 : 0 \to D_1 : 1$, $M_2 : 0 \to D_2 : 1$, $M_2 : 1 \to D_1 : 0$, $M_2 : 1 \to D_1 : 1$, $M_2 : 1 \to D_2 : 1$
e_4	0.14	$M_2 : 0 \to D_1 : 0$, $M_2 : 0 \to D_2 : 0$, $M_2 : 1 \to D_2 : 0$
e_5	2.04	$M_1 : 0 \to D_1 : 0$, $M_1 : 0 \to D_1 : 1$, $M_1 : 1 \to D_1 : 1$, $M_2 : 0 \to D_1 : 1$, $M_2 : 1 \to D_1 : 0$, $M_2 : 1 \to D_1 : 1$
e_6	0.11	$M_1 : 1 \to D_1 : 0$, $M_2 : 0 \to D_1 : 0$
e_7	0.23	$M_1 : 0 \to D_2 : 0$, $M_1 : 1 \to D_2 : 0$, $M_1 : 1 \to D_2 : 1$ $M_2 : 0 \to D_2 : 1$, $M_2 : 1 \to D_2 : 1$
e_8	0.08	$M_1 : 0 \to D_2 : 1$, $M_2 : 0 \to D_2 : 0$, $M_2 : 1 \to D_2 : 0$

Since our goal is to minimize the maximal load, we start to reroute demands, whose paths go through the most loaded edge: e_5. The order in which the algorithm tries to rerout and improve is not specified. Suppose the first attemt is to find an alternative route for demand $M_1 : 0 \to D_1 : 0$. The only choice for an alternative path is to use edges e_2, e_7. The new loads of involved edges are:

e_1	0.23
e_5	1.09
e_2	1.00
e_7	1.18

This is an improvement, therefore, the demand is fixed to the new path. With the new path assignment the most loaded edge is e_7, so wthe heuristic will try to reroute one of the demands going through it. The procedure is repeated until alternative path for all demands on the heaviest edge have beeen considered and no improvement was achieved.

The random heuristic seems to be a good choice if no traffic data is known. We used random as “standard” to compare with other heuristics. The idea for the other heuristics are inspired by well studied bin-packing heuristics. In the case of bin-packing the corresponding fist-fit and best-balance heuristics are known to give reasonable approximations of the optimal solution. Sorted best balance achieves better results than the other heuristics because larger loads are routed first and smaller demands can later be used to balance the load on the edges.

In contrast to the other heuristics k-Opt is guaranteed to produce a locally optimal solution. We observed improvements in almost all cases. The real drawback of this routine is in the running times which easily exeed 1 – 2 hours.

3.3 Software design and implementation

The software system consists of three major modules: network models, input generators, and heuristics, and a module for evaluating diverse combinations of inputs and heuristics – `Eval` as shown in figure 5.

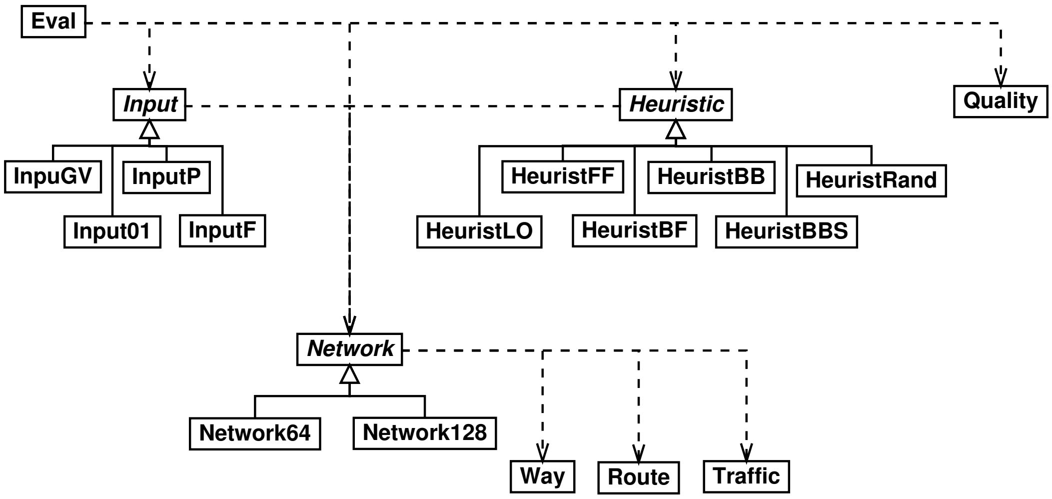

Figure 5: UML-Diagram of the software system

Network models contain the information about the network structure, traffic demands and current routes. They can also compute the lower bound for the maximal load on edges, and write routing information to a file.

- `Network` is an abstract class, which contains the information and functionality common for all network types.
- `Network64` represents a SANbox with 64 ports.
- `Network128` models a 128 type network.

Input generators generate demands for the networks following different strategies.

- `Input` is an abstract class.
- `InputGV` generates uniformly distributed inputs.
- `Input01` produces 1 as an input with a given probability p.
- `InputP` generates a large input (multiplied by given factor f) with probability p.
- `InputF` reads input from file.

Heuristics calculate routes for a given network with it's inputs.

- `Heuristic` is an abstract class, which contains functionality common to all heuristics, such as communication with network.
- `HeuristBF` implements best fit heuristic.
- `HeuristFF` assigns routes according to first fit heuristic.
- `HeuristBB` uses best balance heuristic.
- `HeuristBBS` makes network to sort it's inputs descending first, and then acts like `HeuristBB`.
- `HeuristRand` makes random routing without taking any notion of present condition of the network.

The others are helper classes:

- `Way` contains an information about one possible path,
- `Traffic` contains information about one demand,
- `Route` assigns a path (route) to one demand,
- `Quality` is used by `Eval` and is used to keep information such as lower bound, maximal load, maximal loaded edge.

Neither heuristic modules nor input generators need to know the structure of the network. On the one side it makes it easy to implement new heuristics or input generators, on the other the network models can be replaced with no change needed in heuristic or input generator implementation.

Usage

`assignRoute` takes network type, input type or input file, heuristic and an optional output file as input parameters. It computes the path assignment and writes it to output file, if specialized.

The syntax is following:
`assignRoute` *network input heuristic* [`o` *outfile*]
where

> *network* is `64` or `128`, for type 64 and 128 networks respectively;
>
> *input* is one of
>
> > `i` *file* for reading input from *file* (`InputF`),
> > `u` for uniformly distributed input (`InputGV`),
> > `p` n for input generator producing 1 with probability $1/n$ (`Input01`),
> > `n` n `f` *factor* for producing large by factor *factor* inputs with probability $1/n$ (`InputP`);
>
> *heuristic* is one of
>
> > `ff` for First Fit,
> > `bf` for Best Fit,
> > `bb` for Best Balance,
> > `bs` for Best Balance with sorted inputs,
> > `rand` for Random
> > `k` n for k-Opt with $k = n$.

We also implemented some test programs to gain the statistic data:
`stat` n *network inputs*
where n is number of experiments, *network* as above and *inputs* is one or more inputs as above. The program produces n inputs of each type, finds path assignments with all but k-Opt heur istics and gives back average lower bound and greatest last values for each input-heuristic pair.

Another test program we implemented is `optStat`:
`optStat` n *heuristic network inputs* `k` k
with n, *network*, *inputs* as above, *heuristic* as in `assignRoute` except for k-Opt, and k is the parameter for the k-Opt heuristic. This program computes first path assignment with given *heuristic* and tries to optimize it with k-Opt. The result is comparison between *heuristic* and k-Opt over n experiments.

3.4 Integer program model

The problem model as described in section 2.1 can be rewritten as an integer program. The integer program consists of the following linear equations and inequalities (e.g. for Type 128 network):

$$\begin{aligned}
&\text{Let } P = \{1, \ldots, 128\} \text{ be set of ports,} && (1)\\
&S = \{1, \ldots, 8\} \text{ set of intermediate switches} && (2)\\
&\forall i, j \in P,\ s \in S \quad x_{i,j}^{(s)} \in \{0, 1\} && (3)\\
&\forall i, j \in P \quad \sum_{s \in S} x_{i,j}^{(s)} = 1 && (4)\\
&\forall \text{ edges } (i, s) \quad \sum_{k \in P} x_{i,k}^{(s)} r_{i,k} + \sum_{k \in P} x_{k,i}^{(s)} r_{k,i} \le B && (5)\\
&\text{minimize } B && (6)
\end{aligned}$$

where B is the maximal load on the network and

$$x_{i,j}^{(s)} = \begin{cases} 1, & \text{if demand } r_{i,j} \text{ is routed over switch } s \\ 0, & \text{else} \end{cases}$$

The difference between integer program and a linear program is that in integer program solution the x variables can only have integer values: 0 or 1, in a linear program solution they may also assume floating point values. The equation (3) would be like

$$\forall i, j \in P,\ s \in S \quad x_{i,j}^{(s)} \in [0, 1]$$

meaning that portion of the demand $r_{i,j}$ is routed over switch s. This condition is a relaxation of that in equation (3).

The next equation (4) just says that exactly the demand $r_{i,j}$ is routed between ports i, j over all switches s. For the integer program it means that for one and only one s $x_{i,j}^{(s)} = 1$.

Equation (5) sets the upper bound B for loads on all edges. It means that the load sum on each edge should be less than B.

With object to minimize B (6) we express the wish to keep maximal load minimal. The values of x-variables for the found solution describe the optimal routing.

For other network types parameter in (1) and (2) have to be changed.

Though integer programming is a NP-complete problem, it is also a very important one. Many people have done a lot of research and work in that area in order to make good and fast solvers. As result there are solvers for integer programming available that are relative fast. The one we used is CPLEX[1].

The model was described in A Modeling Language For Mathematical Programming (AMPL[2]), and the results in section 4 were obtained using CPLEX solver.

4 Results

4.1 Statistical evaluation

The following tables show some computational results. Each row of the table compares the performance of the five algorithms on inputs generated by the input generator specified in the first column. The numbers characterize the average performance over ten inputs of that type.

> The entries of the table have been obtained as the ratio of the load produced by the heuristic over the lower bound on the load.

We exemplify this with a concrete instance and show how the bold value **1.24528** in the table was generated. The input generating algorithm 1 with probability 1/64 produced a sequence $R_1, R_2, \ldots, R_{10}$ of inputs. For each R_i the lower bound $\rho^*(R_i)$ was calculated. The best-fit heuristic was run on input R_i, the load of this assignment is denoted $\Lambda(BF(R_i))$. The average performance was finally obtained as:

$$\frac{1}{10} \sum_{i=1}^{10} \frac{\Lambda(BF(R_i))}{\rho^*(R_i)}$$

[1]CPLEX is linear, mixed-integer and quadratic programming solver. The maintainer is CPLEX division of ILOG [5].

[2]AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables, developed at Bell Laboratories.

AMPL lets one use common notation to formulate optimization models and examine solutions, while the computer manages communication with an appropriate solver.Successfully used in demanding model applications around the world, AMPL is available on a variety of PC and UNIX platforms. More information about AMPL in [4].

Table 1: Type 64 network

	FF	BF	BB	SoBB	RAND
Uniformly distributed					
	1.06522	1.09076	1.00864	1.00004	1.21321
1 with probability					
1/4	1.02834	1.07105	1.01191	1.01191	1.31417
1/16	1.05417	1.08931	1.04246	1.04246	1.67496
1/64	1.24528	**1.24528**	1.24528	1.24528	2.15094
1/512	2.28571	2.28571	2.28571	2.28571	4.57143
1/1024	3.33333	3.33333	3.33333	3.33333	5.33333
Factor 100 with probability					
1/4	1.05975	1.09067	1.02673	1.00001	1.40567
1/16	1.10331	1.18113	1.08668	1.00001	1.67364
1/64	1.08718	1.39074	1.24096	1.00001	1.94209
1/512	1.22016	1.74462	1.61577	1.11816	2.31087
1/1024	1.20426	1.79604	1.55409	1.12589	2.02765
Factor 500 with probability					
1/4	1.05517	1.07962	1.02690	1.00001	1.41476
1/16	1.09232	1.18616	1.09193	1.00001	1.73884
1/64	1.25942	1.49942	1.28602	1.02441	2.20444
1/512	2.37386	2.89089	2.54823	2.37386	3.88258
1/1024	2.81228	3.35943	3.06592	2.81064	3.89611
Factor 1000 with probability					
1/4	1.05759	1.08367	1.02655	1.00001	1.41567
1/16	1.10007	1.19820	1.08802	1.00004	1.74687
1/64	1.25653	1.51018	1.30433	1.05419	2.24880
1/512	2.77952	3.24661	2.87444	2.77952	4.38120
1/1024	3.60928	4.07431	3.77362	3.60822	4.75084
Factor 4000 with probability					
1/4	1.05582	1.08004	1.02628	1.00016	1.41654
1/16	1.10624	1.19695	1.09135	1.00339	1.75301
1/64	1.29337	1.51415	1.31629	1.08357	2.28438
1/512	3.18812	3.54008	3.21534	3.18812	4.88343
1/1024	4.60227	5.03404	4.65499	4.60193	5.81576

Table 2: Type 128 network

	FF	BF	BB	SoBB	RAND
Uniformly distributed					
	1.05236	1.05868	1.00297	1.00001	1.17783
1 with probability					
1/4	1.03779	1.05025	1.00662	1.00662	1.29801
1/32	1.11675	1.13706	1.07614	1.07614	1.82741
1/128	1.23077	1.23077	1.15837	1.15837	2.17195
1/512	1.55556	1.55556	1.33333	1.33333	3.33333
1/1024	1.63265	1.63265	1.63265	1.63265	3.91837
Factor 50 with probability					
1/4	1.04403	1.05490	1.01146	1.00001	1.28594
1/32	1.07848	1.12688	1.06490	1.00001	1.55210
1/128	1.11492	1.21692	1.10337	1.00001	1.59536
1/512	1.15660	1.26020	1.14894	1.00001	1.52081
1/1024	1.20188	1.27930	1.16499	1.00001	1.45033
Factor 100 with probability					
1/4	1.04310	1.05765	1.01181	1.00000	1.29532
1/32	1.08936	1.13835	1.07832	1.00001	1.64909
1/128	1.13152	1.28187	1.16078	1.00001	1.86923
1/512	1.15259	1.44978	1.27817	1.00001	1.86658
1/1024	1.17123	1.50721	1.34991	1.00001	1.79037
Factor 500 with probability					
1/4	1.04122	1.05470	1.01190	1.00001	1.29933
1/32	1.08401	1.16855	1.08561	1.00001	1.75356
1/128	1.21807	1.48044	1.25240	1.00001	2.36941
1/512	1.37163	2.00984	1.60833	1.27582	2.92033
1/1024	1.70248	2.33018	2.03912	1.68174	3.09234
Factor 1000 with probability					
1/4	1.04167	1.05821	1.01222	1.00000	1.30084
1/32	1.08393	1.16316	1.09296	1.00032	1.77014
1/128	1.31577	1.53745	1.27524	1.01042	2.46896
1/1024	2.12773	2.69518	2.34948	2.12773	3.64449

During the first phase of the project we where experimenting with a special network. Basically the network is like Typre 64, the difference is that instead of double it has single connections between switches. This network was used to analyze several input strategies and routing heuristics. The routing problem for this network is considerably smaller than the problem for the Type 64 network. This makes a great difference in the computation time for the integer program and for the k-Opt heuristic. Though we implemented these algorithms for general networks we have evaluated them only for the simplified Type 64 network. The reason for this restriction was that only with this network the number of experiments needed for the statistical evaluation could be performed in reasonable time .

Table 3: Simplified Type 64 network

	FF	BF	BB	SoBB	RAND	OPT	1-OPT
Uniformly distributed							
	1.05777	1.08337	1.00287	1.00002	1.10636	1.00000	1.00002
Factor 100 with probability							
1/2	1.04720	1.05972	1.00475	1.00003	1.14636	1.00001	1.00002
1/16	1.03714	1.09033	1.03226	1.00010	1.33402	1.00001	1.00004
1/32	1.05344	1.13930	1.05187	1.00001	1.47163	1.00001	1.00001
1/128	1.04069	1.22027	1.14611	1.00001	1.77608	1.00001	1.00001
1/512	1.49359	1.98240	1.60244	1.42079	2.97277	1.18329	1.54706
1/1024	1.05035	1.27032	1.16791	1.00001	1.61586	1.00000	1.00002
Factor 1000 with probability							
1/2	1.04667	1.05744	1.00477	1.00001	1.14731	1.00000	1.00000
1/16	1.03777	1.10549	1.03541	1.00000	1.37344	1.00000	1.00000
1/32	1.06219	1.16059	1.07659	1.00130	1.56908	1.00002	1.00816
1/128	1.16824	1.44765	1.22576	1.03473	2.25696	1.02085	1.10278
1/512	1.52917	1.94573	1.64528	1.45989	2.30137	1.22531	1.42384
1/1024	1.85392	2.36375	2.02100	1.82638	2.50154	1.70653	1.82162

The results differ from those obtained in first phase of the project (compare Table 3 and Table 4). In the first phase of the project we were assuming that all the demand would be between ports located at some machine switch M_s and ports located at some disk switch D_u. This assumption reduces the size of the input matrix from 64×64 to 32×32. networks in the first stage were assumed to be directed with only traffic between machine and disk switches. Thus the input matrix had dimensions 32×32 instead of 64×64 which it has now. Thais means that 4 times more demands have to be routed over the same set of available edges. Therefore with the larger input matrix the edge loads can be balanced much better and the ratio to the lower bound is smaller.

Table 4: Simplified Type 64 network with 32×32 input.

	FF	BF	BB	SoBB	RAND
Uniformly distributed					
	1.00401	1.02783	1.00915	1.00015	1.22408
Factor 1000 with probability					
1/2	1.00924	1.02752	1.01778	1.00005	1.28737
1/16	1.12357	1.13486	1.12443	1.01555	1.68599
1/32	1.23210	1.23216	1.23214	1.06034	1.93165
1/128	1.74524	1.76007	1.76051	1.69249	2.56612
1/512	2.54587	2.55284	2.59625	2.50725	3.05256
1/1024	2.93044	2.93992	2.99362	2.89226	3.17847

The data shown in the tables shows that it is reasonable to rank the algorithms in the following order RAND < BF < FF < BB < SoBB $\leq$ k-Opt. It is also obvious that some inputs are more difficult than others. The results also confirm our consideration from section 3.1 that the more (similar) demands an input contains, the easier it is to find a good path assignment.

4.2 Computation time

The average computation time for all heuristics (excluding k-Opt) for Type 64 networks is 15 sec. The same heuristics compute path allocation for Type 128 network within average 1 min. 1-Opt needs about 2.8 min. for Type 64 network and 16 min. for Type 128 network.

The integer program computes the optimal route for Type 64 networks in average in 1.59 hours. Whereby the computation time of an integer program varies greatly: from 11.6 seconds to 28 hours. The average computation time for Type 128 network is 39 hours.

Heuristics computation time was measured on a Celeron 433 MHz machine with 128 Mb RAM. The integer program was tested on an UltraSpark 2 with two 300 MHz processors with 768 Mb RAM.

5 New Problem Definition

In the third phase of the project we made some research on the new problem definition: The matrix R as defined in section 2.1 is unknown. Instead we assume to know

- the current path assignments $P(i, j),\ \forall i, j \in$outer ports,
- the total incoming and outgoing traffic $F(i), T(i)$ through outer port $i,\ \forall i$,
- if the intermediate switch S_i needs to buffer, the buffering information,

Task is to find an improved path assignment which reduces the need for buffering.

The research was concentrated on reconstruction of demand matrix R. The idea was: since we already have good path allocation algorithms, we need to get good approximation for input matrix R and find new path assignment by applying formerly implemented algorithms.

The first observation one can make is that $F(i), T(i)$ are the row and column sums of the matrix R. The general problem of matrix reconstruction from it's row and column sums does not have unique solution. Though in our case we have further conditions that restrict the set of acceptable solutions, there still can be more than one solution.

The corresponding mathematical model is following:

Let

$P(i, j) = (e_x, e_y)$ is Path assignment over edges e_x, e_y for traffic from port i to port j,

$n =$ number of outer ports (64 or 128),

$\gamma =$ capacity of edges e, since all connections apply the same technology the capacities should be equal,

$\rho(e) =$ buffering on edge e,

$r_{i,j} \in R$ traffic demand from port i to port j,

Then

$$0 \leq r_{i,j} \leq \min\{F(i), T(j)\} \tag{7}$$

$$\sum_{j=1}^{n} r_{i,j} = F(i) \ \forall i = 1 \dots n \tag{8}$$

$$\sum_{i=1}^{n} r_{i,j} = T(j) \ \forall j = 1 \dots n \tag{9}$$

$$\sum_{e \in P(i,j)} r_{i,j} = \gamma + \rho(e) \ \forall e \text{ with } \rho(e) > 0 \tag{10}$$

$$\sum_{e \in P(i,j)} r_{i,j} \leq \gamma \ \forall e \text{ with } \rho(e) = 0 \tag{11}$$

describe a linear program model for the problem. This time we use linear and not integer programming because the values of demands do not have to be integer values, and there are no other conditions that require integer solution, as there were in initial problem.

Inequality (7) sets the natural bounds for each demand, saying that it can not be negative or greater than sum of all demands going from it's source port or coming into it's destination port.

Equations (8) and (9) express that $F(i)$ is sum of demands with source i and $T(j)$ is total of demands with destination j. Equation (10) states the buffering information and inequality (11) sets the upper bound γ for sum of demands routed over connection e with no buffering.

Since there is more than one solution possible, we can not just say 'find R'. However we could solve linear problems with model above and object: 'minimize $r_{i,j}$' for all values of i and j. That would provide us a matrix $\underline{R}$ with minimal values for each $r_{i,j}$. The matrix $\underline{R}$ it self is not necessarily a possible solution.

Analog we could obtain matrix $\overline{R}$ by solving the maximization problem for each $r_{i,j}$. That would be $2 \times n^2$ linear problems. Though both matrices $\underline{R}$ and $\overline{R}$ do not necessarily belong to possible solutions set, the give us a bounding box for that set.

Another idea was to consider convex combinations of solutions. We coud not proceed further from this point simply because the funding for the project terminated.

Bibliography

[1] http://www.qlogic.com

[2] http://www.spacey.net/ldavis/Design Connector FibreChannel.html

[3] http://www.fibrechannel.org/technology/

[4] http://www.ampl.com

[5] http://www.cplex.com