A Genetic Algorithm for Designing Parallel Processor Interconnection Networks

Morris Marden

Abstract

Parallel processor interconnection networks traditionally have been designed from a known set of topologies based on geometric shapes, called conceptual topologies. However, conceptual topologies only explore a very small portion of the solution space, so they may not produce a very desirable solution. Instead, we propose using a genetic algorithm to design these networks. Our results show that such an algorithm can design lower latency networks than conceptual topologies, using fewer links. In particular, our algorithm, called GAPPIN, can design a network with four links per node that has an average latency that is 24% lower and a maximum latency that is 38% lower than a two-dimensional torus. Furthermore, GAPPIN designed a network using six links per node that has an average latency that is 19% lower and a maximum latency that is 33% lower than a three-dimensional torus. Our algorithm can also model the affects of contention in the network. To ensure that it is practical to implement the networks GAPPIN designs, we modified the algorithm to produce networks in which none of the wires are longer than a maximum length. We show that GAPPIN can create designs for networks in which conceptual topologies fail this constraint. Finally, GAPPIN can also design networks that are optimized for fault tolerance.

Keywords:genetic algorithm, interconnection network, topology, CAD, network modelling

Abstract

Parallel processor interconnection networks traditionally have been designed from a known set of topologies based on geometric shapes, called conceptual topologies. However, conceptual topologies only explore a very small portion of the solution space, so they may not produce a very desirable solution. Instead, we propose using a genetic algorithm to design these networks. Our results show that such an algorithm can design lower latency networks than conceptual topologies, using fewer links. In particular, our algorithm, called GAPPIN, can design a network with four links per node that has an average latency that is 24% lower and a maximum latency that is 38% lower than a two-dimensional torus. Furthermore, GAPPIN designed a network using six links per node that has an average latency that is 19% lower and a maximum latency that is 33% lower than a three-dimensional torus. Our algorithm can also model the affects of contention in the network. To ensure that it is practical to implement the networks GAPPIN designs, we modified the algorithm to produce networks in which none of the wires are longer than a maximum length. We show that GAPPIN can create designs for networks in which conceptual topologies fail this constraint. Finally, GAPPIN can also design networks that are optimized for fault tolerance.

1. Introduction

As processors continue to become faster and more machines are built using a relatively large number of processors, buses are becoming less able to provide the required bandwidth for such machines. The solution is to use interconnection networks instead of buses. Traditionally, there have been two methods used to design these networks: brute force enumeration and conceptual topologies. Under brute force enumeration, one tries all possible combinations of connecting the nodes together in the network to find the best solution. This is not feasible to do for a relatively large number of processors, as the size of the problem is exponential in terms of the number of nodes in the network and the number of links per node. Therefore, engineers usually use a common set of topologies that are based on geometric shapes to design interconnection networks. We call these topologies "conceptual topologies." Some examples of conceptual topologies in research and commercial machines are rings (which are used in the SCI protocol [6] and in Data Scalar architectures[4]), trees (used in the Thinking Machines CM-5 [5]), hypercubes (used in the SGI Origin [3]), and multidimensional toruses (used in the Cray T3D and T3E [2]).

The problem with these two approaches is that they use two different extremes to search the solution space for designing interconnection networks. The brute force approach looks at the entire solution space, which is simply too large to practically do for reasonably sized systems. Conceptual topologies, on the other hand, only cover a very small portion of the solution space, so one may overlook some more desirable solutions. Therefore, we suggest using an algorithm that intelligently searches a large portion of the solution space without enumerating all possible solutions. We do this using a genetic algorithm, which we call the Genetic Algorithm for Parallel Processing Interconnection Networks (GAPPIN).

Genetic algorithms are commonly used to solve a variety of problems. In 1958, Friedberg created the first genetic algorithm, which attempted to make a computer learn by mutating small FORTRAN programs [8, 9]. Today, genetic algorithms are commonly used for VLSI CAD design [10-12]. Genetic algorithms have also been used before in computer architecture. For example,

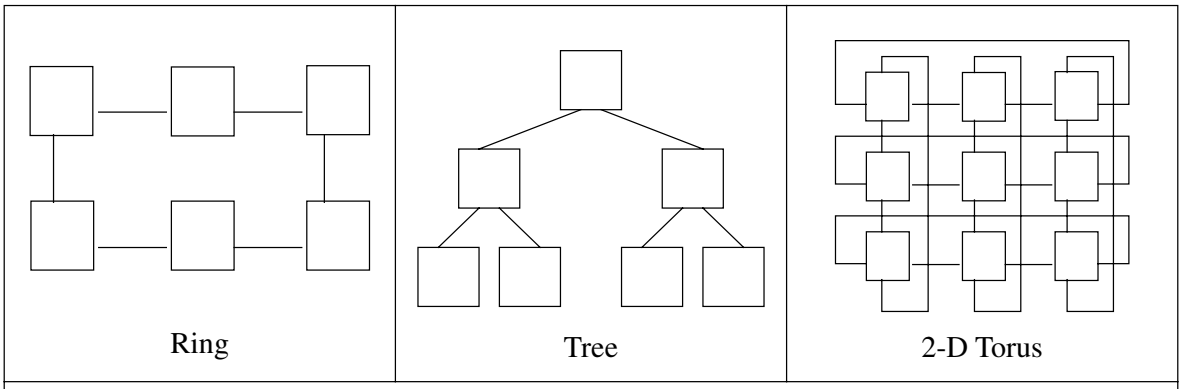


Figure 1: Examples of conceptual topologies

Emer and Gloy studied using genetic algorithms for branch predictors [7]. Genetic algorithms search the solution space by starting with an initial set of solutions, modifying these solutions, and determining if the new solutions are better than the previous generation of solutions. Usually, the solutions are modified by picking two solutions at random (the parent solutions) and combining them in some fashion to create a new solution (the child solution). The algorithm then evaluates the costs of the new sets of solutions, which is called the fitness value of the solutions. If the fitness value is worse than a threshold value (often the median or average value), then the solution is discarded. After a certain number of iterations (called generations) have occurred, the algorithm reports the best solution.

In the past, Lau and Tsang have studied the use of a genetic algorithm to design parallel processor interconnection networks [1]. Their algorithm, however, misses many important issues in optimizing the network. First, their algorithm only minimizes for the maximum latency in the network. In contrast, our algorithm optimizes for average latency in addition to maximum latency. We have also included design constraints in our algorithm. The current version of our algorithm can design networks such that no wires are longer than a maximum length (which will be discussed later in this paper). Second, their algorithm creates a new generation by only performing mutations on the current generation. Instead, we believe that one would get much better results by both combining solutions and mutating solutions to create a new generation. Lastly, Lau and Tsang work only compares their algorithm to other genetic algorithms that can be used. We believe that it is important to compare the results of a genetic algorithm to current methods.

The first version of our algorithm designed networks that were optimized for performance alone, ignoring constraints of building such a network. Section 2 describes this version of the algorithm and section 3 shows the results of this algorithm. In section 4, we describe a modification to the algorithm so that it will be able to model the effects of contention on the network. Since many of the networks that the algorithm designed appeared to be difficult to build, we modified the algorithm to prevent any links in the network from being longer than a maximum length. We describe this modification and its results in section 5. Fault tolerance of large networks is becoming an important issue, so in section 6 we discuss modifications to our algorithm to include fault tolerance. Finally, we conclude with some remarks in section 7.

2. Algorithm for Designing Networks for Performance

2.1 Network Model

GAPPIN models a directly connected network, since it would be much more difficult to create an algorithm to design networks with an arbitrary number of switches. Note that this network model is not inherent in the algorithm itself and can be replaced by other models without affecting our algorithm's basic functionality. All of the links in our network model have the same bandwidth and latency, which is common in interconnetion networks. Messages in the network are all the same length and take a single unit of time to traverse a link in the network. This is the case of packet switched networks, where packets are equal in size and can be considered as messages. The links in the network are bidirectional and can carry a maximum of one message per direction per unit of time. Messages traveling in opposite directions along the same link do not affect one another. A network can implement this using separate wires and queues for each direction of the link. Finally, each node in the system has the same number of links, which is finite in number and specified by the user. This is the case in most networks, since most networks use the same components for each of the nodes in the network (to have a different number of links per node requires one to use different components in the network).

We made several simplifications to our network model to avoid many implementation-specific details and to reduce the complexity of the algorithm. First, our model specifies that messages are uniformly distributed across the network (i.e., it is equally likely that a message will be sent between any pairs of nodes in the network). To separate the effects of the topology of the network from the effects of the routing algorithm, we statically route messages in the network using shortest path routing. Since designing the network topology and the routing algorithm are equally difficult problems to solve, we suggest separating these two problems and using a similar algorithm for designing the routing algorithm after designing the network topology. Lastly, we currently do not consider the effects of contention in the network.

2.2 Creating the Initial Population

When our algorithm starts, it creates an initial set of solutions for the interconnection network. A solution can be viewed as a proposal for the final design of the network and a set of solutions is called a population. We create the initial population by first making two lists of nodes: a list of nodes currently connected in the network and a list of nodes not connected in the network. The list of connected nodes initially only holds the first node in the network, which is the seed of the network, and the rest of the nodes are in the list of unconnected nodes. We then randomly connect a node from each list to each other, moving the node from the list of unconnected nodes to the list of connected nodes, until the list of unconnected nodes is empty. This ensures that none of the solutions will be a disconnected network, which is a network in which there is no path to one or more nodes in the network. The remaining links in the network are created by connecting them to random nodes.

2.3 Creating New Generations

In each iteration, our algorithm creates a new generation of solutions. The new generation of solutions starts out twice as large as the previous generation of solutions. Half of the solutions in the new generation are the previous generation of solutions and the other half are new solutions. This prevents the design of the network from degrading if the new solutions for the network have lower performance than the previous generation of solutions.

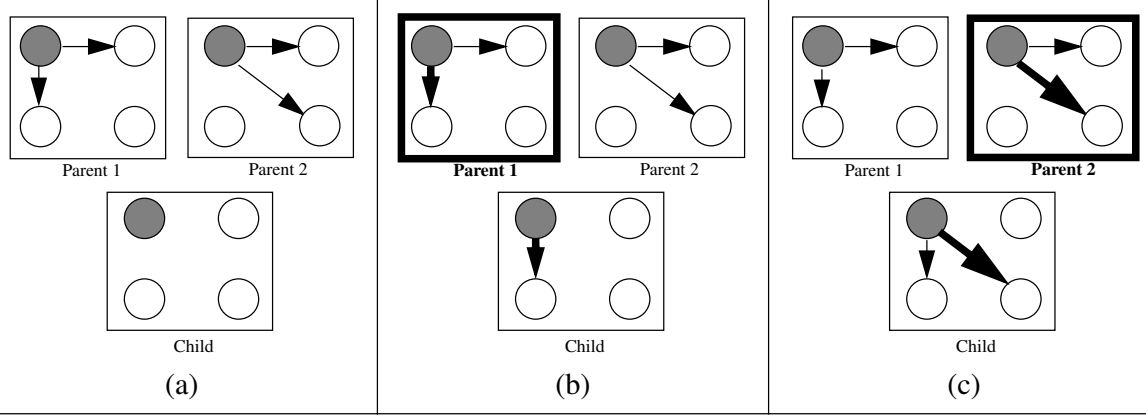


Figure 2: Example of creating a new solution using inheritance (shaded node is the node that GAPPIN is currently working with)
(a) At beginning, no links in child yet
(b) Adding link from 1st parent to child
(c) Adding link from 2nd parent to child, now finished with node in child

To create a new solution, our algorithm randomly picks two solutions to mate together to create a new child solution. GAPPIN then creates links for each node in the child solution using two mechanisms: inheritance and mutation. Note that these mechanisms work differently in this algorithm than in a pure genetic algorithm. Inheritance creates a link for a node by randomly picking a parent to inherit from. Next, our algorithm randomly picks a link from the same node in the parent network and copies the link into the child solution. Figure 2 shows an example of creating a new solution using inheritance for the case of 4 nodes with 2 links per node. Figure 2a shows the solutions before any links have been created. In the next step (2b), GAPPIN acquired a link from the first parent and copied this link to the child. For the next link (in part 2c), our algorithm chose a link from the second parent and copied the link to the child. At this point, our algorithm has finished creating the links for the first node in the child and would continue this process to create links for the rest of the nodes. This example shows that while the child solution has characteristics of both its parents, it is different from both of them.

Mutation, which is the second method for creating links in the child solution, creates a link by randomly picking a node to link to. Mutation helps improve the performance of the network by adding variation to the generation of solutions. Without mutation, each generation is limited to being similar to the previous generation of solutions. Our algorithm uses inheritance much more often than mutation, since inheritance has memory (i.e., maintains the good characteristics of pre-

vious solutions) and mutation is memory-less (i.e., is not based on characteristics of previous solutions).

After creating the new generation, GAPPIN computes the cost of each of the new solutions, which is also known as the fitness value. Our algorithm then sorts the solutions of the new generation by their cost. Solutions with a cost less than the median cost survive through the generation and solutions with a cost greater than the median cost are removed from the generation.

$$\text{Cost} = \alpha \times \text{Average latency} + \beta \times \text{Maximum latency}$$

Where α and β are the weights of the individual components of the cost, which are set by the user.

The cost of a network is broken up into two components: average and maximum latencies. We define latency as the number of hops from the source to the destination node. For these latencies, the messages will use the shortest path to route from the source to the destination, since we are not modelling contention, so there is no benefit to taking a longer path than necessary. To calculate the latencies, we use Dijkstra's algorithm [20] to find the shortest paths between all pairs of nodes in the network.

As mentioned before, our algorithm is not a pure genetic algorithm. If we used a pure genetic algorithm, the algorithm would first convert the configurations of the parent networks to string representations. Next, the algorithm would randomly choose how many links to get from the first parent. Finally, it would get links for the child from the first part of the first parent and the rest of the links would be from the second parent. Instead, we believe that by randomly choosing the links from each parent, the algorithm will search more of the solution space and is less likely to get trapped in a local minima.

2.4 User Defined Variables

Our algorithm includes five different variables, which the user can change to affect the manner in which GAPPIN designs networks. These variables are: weights of the terms in the cost function, population size of each generation, number of generations to create, probability to use mutation to create links instead of inheritance, and probability to accept redundant links. First, the weights of the terms in the cost function allow the user to specify the relative importance of the measurements of the cost of a network. For example, in our experiments, we specified the weighting of the terms of the cost function as 1.0 (i.e., average and maximum latencies), which means that each of the terms are equally important. The population size of each generation and the number of generations to create effectively control how long the algorithm will run. The larger these two values are, the larger the portion of the solution space is explored, resulting in a better final design. However, as the value of these variables increase, the time that it takes to create the final design also increases. The probability to use mutation to create links rather than inheritance allows the user to specify the probability that our algorithm uses mutation instead of inheritance to create a link for the network.

Finally, the user can set the probability that the algorithm accepts redundant links when it is designing the network. Redundant links occur when two nodes in the network have more than one direct link to each other. This may be helpful when there is contention in the network, since it can increase the bandwidth between a pair of nodes in the network that have heavy traffic. For

example, fat trees introduce redundant links to reduce the bottlenecks in tree topologies. However, the more redundant links there are in a network, the more likely the network is a disconnected network. Disconnected networks are trivially removed from the generation of solutions by the cost function, since they have infinite latencies. Disconnected networks occur more frequently with more redundant links, since the redundant links can use up available links that are needed to connect to other nodes. Evaluating disconnected networks wastes time, since they are trivially rejected and the time used to create them could have been used to create valid networks.

3. Results

3.1 Graphical Representation of Results

Figure 3 shows the results of our algorithm for designing eight, twelve, and sixteen node systems with three links per node. The twelve and sixteen node systems are typical of the networks that GAPPIN designs, in that there does not appear to be any particular geometric shape to the topologies. However, in the case of the eight node systems, GAPPIN designs a network with a topology that is similar to a hypercube, but changes two of the links. This network has higher performance than a standard hypercube (8% lower average latency and 33% lower maximum latency), yet is not much more difficult to build than a hypercube. It is interesting that GAPPIN designed a network similar to a conceptual topology for eight nodes, but not for larger systems. The likely reason for this is that with only eight nodes, there are few ways to design the network.

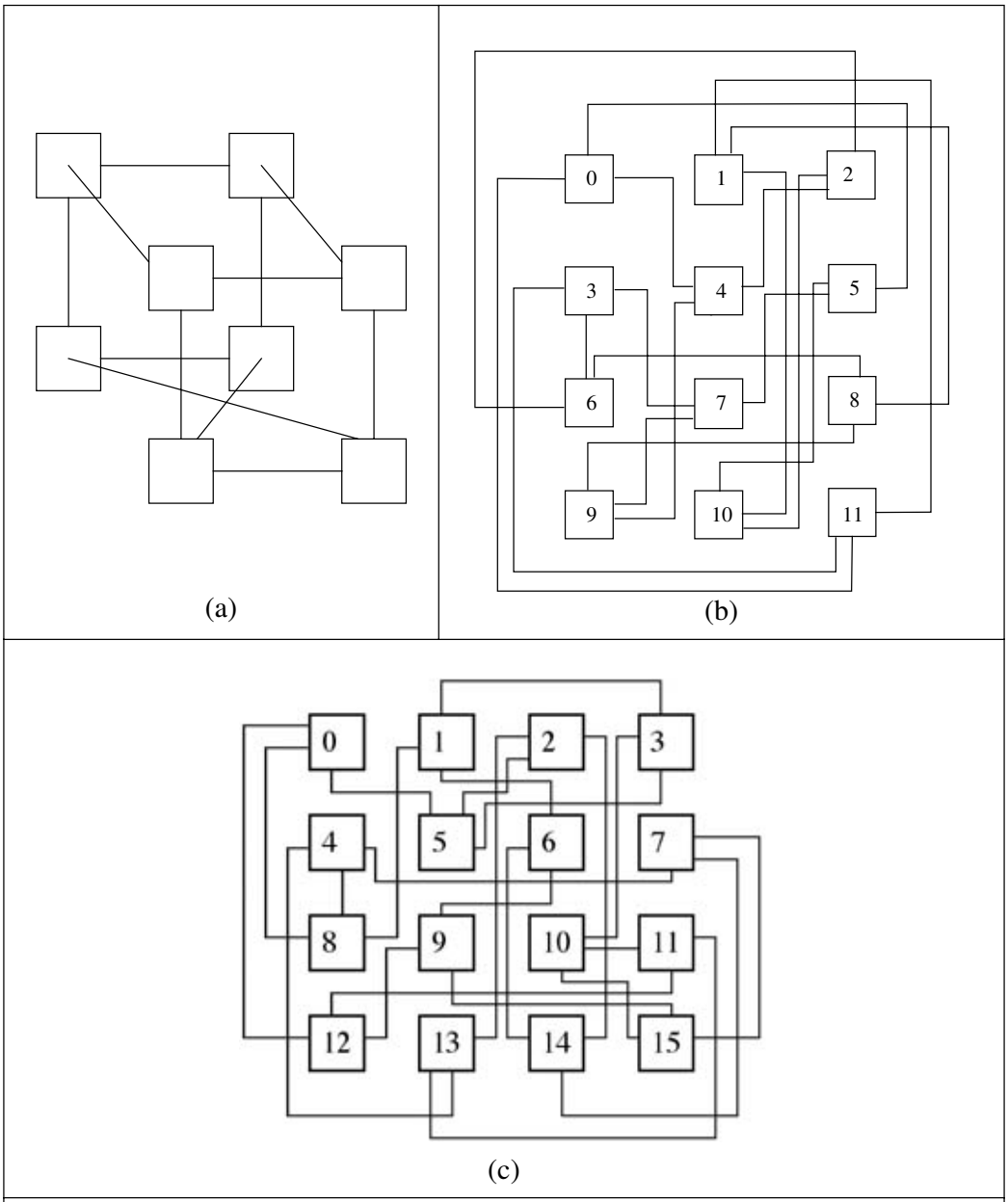


Figure 3: Graphical representatioicpp.psn of results with 3 links per node
(a) results for 8 nodes, (b) results for 12 nodes, (c) results for 16 nodes.

3.2 Comparison of Results to Conceptual Topologies

Name of Topology	# of links/node	Average Latency	Maximum Latency
Ring	2	16.00	32
Tree *(63 nodes)*	3 *(but not all used)*	6.68	10
2-D Torus	4	4.00	8
3-D Torus	6	3.00	6
GAPPIN	3	3.94	7
GAPPIN	4	3.03	5
GAPPIN	5	2.67	4
GAPPIN	6	2.43	4

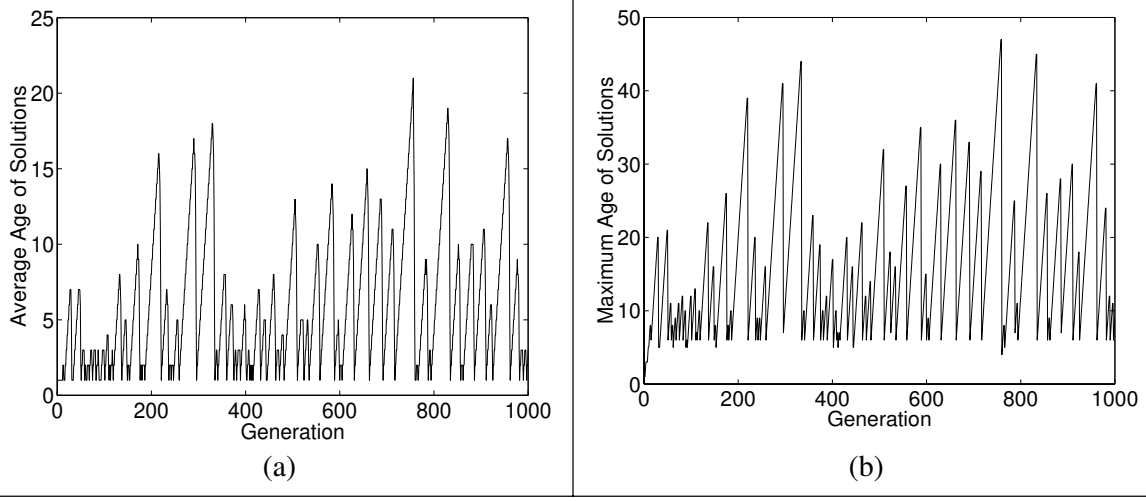
Table 1: Comparison of latencies for conceptual topologies and results from GAPPIN, for systems with 64 nodes

To measure how well our algorithm works, we compared the results of the algorithm to conceptual topologies, which are the current accepted method of designing interconnection networks. Table 1 compares the average and maximum latency results of GAPPIN for a 64 node system to four commonly used conceptual topologies. The latencies in these results are reported from GAPPIN, as described in section 2.3. To create these networks, we specified that the cost function should consider the latencies of the network equally (i.e., the weight of each of the components was set to 1.0). Note that for the tree, we placed nodes at each of the levels of the tree and not just the leaves, since this matches the network model better and results in lower latencies.

As one can see, the resulting designs from the algorithm outperform the conceptual topology designs. These results demonstrate that the network designed by the algorithm that uses three links per node has higher performance than the two dimensional torus network, which uses four links per node. Likewise, the network that the algorithm designed using four links per node has better performance than the three dimensional torus design, which uses six links per node. Since ports on nodes for links are often expensive, this potentially can yield a large cost savings. Alternatively, one can use the same number of links one would normally use for a conceptual network and instead have a much higher performance network.

3.3 Analysis of GAPPIN Over Time

In this section, we analyze the behavior of our algorithm over time as it is designing a network. This way, we can better understand GAPPIN.



(a) (b)

Figure 4: Average and maximum age of the solutions over time for 512 nodes and 12 links per node

(a) average age, (b) maximum age

Figure 4 shows the average and maximum age of the solutions (i.e., how many generations each solution has survived) over time when our algorithm was designing the network. None of the solutions lasts for a long period of time in our algorithm, which means that GAPPIN continues to improve the design over time. When GAPPIN comes up with a better solution, we believe that it replaces the old solutions with variants of the new one. This accounts for the large spikes in the ages. When our algorithm discovers a new, better solution, the average age drops close to zero, but the maximum age remains well above zero. This means that it takes several generations for the new best solution to affect the entire generation of solutions.

4. Contention Modelling

4.1 Modifications to the Algorithm to Model Contention

To model contention, we need to make some changes to the network model. First, we model that the network performs dynamic routing with perfect knowledge of what will be the lowest latency path at the time that the message crosses the network. While real dynamic routing algorithms cannot have perfect knowledge of the best path in the network at any moment in time, they can use global knowledge to guide such routing decisions. For example, Thottethodi et al have proposed a dynamic routing algorithm that collects global information about contention in the network to make decisions about message transmissions [26]. We model moderate contention in the network as each node in turn sending a message to every other node in the system. Messages starting from the same node in the network do not conflict with each other, but messages staring from different nodes do conflict with each other. As mentioned before, messages traveling opposite directions along a link do not conflict with each other, but messages traveling the same direction across a link do conflict with each other. The network then dynamically routes messages around congested links.

We define the latencies under moderate contention as the hop count from the source to the destination and does not include the length of time that a message needs to stall in the network. The path that messages take may not be the shortest path, because a longer path may take less time to travel along than the shorter path. Thus, the average and maximum latencies under moderate contention give us a measure of the length of alternate paths for messages in the network. If there are no alternate paths between nodes (for example, as in a tree topology), then these latencies will be the same as in the cases under no contention.

In addition, our algorithm also calculates the standard deviation in the number of messages that are sent along each link in the network under moderate contention. This measurement tells us whether some links in the network are more heavily used than other links in the network, causing these links to become bottlenecks in the network. For example, in a tree topology, there is a large variance in the number of messages along each link of the network, since more messages need to travel through the links near the root of the tree than in the links near the leaves. Therefore, the links at the top of the network are the bottleneck of the tree network. Ideally, all of the links in the network would be equally used, which would correspond to a standard deviation of zero.

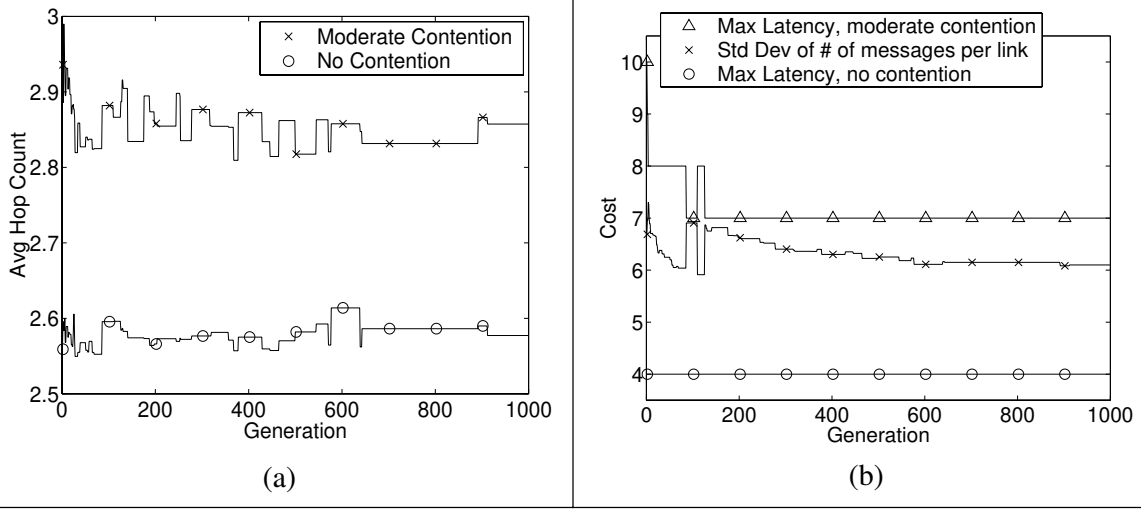
4.2 Results from Modelling Contention

Name of Topology	# of links/node	Average Latency, some contention	Maximum Latency, some contention	Standard deviation of # of messages/link
Ring	2	18.16	61	103.19
Tree *(63 nodes)*	3 *(not all used)*	6.48	10	104.92
2-D Torus	4	4.48	12	12.79
3-D Torus	6	3.36	10	8.50
GAPPIN	3	4.05	9	9.58
GAPPIN	4	3.23	8	5.96
GAPPIN	5	2.93	7	4.38
GAPPIN	6	2.64	6	3.91

Table 2: Comparison of costs under moderate contention for conceptual topologies and results from GAPPIN, for a system with 64 nodes

Table 2 compares the results under moderate contention of the networks designed by the algorithm to the four conceptual topologies for results. Note that for the ring, the contention must have been clustered in one section of the network, causing some messages to take the long route from the source to destination. This is an artifact of our dynamic routing algorithm, which attempts to route around congested links. Not surprisingly, the tree topology has a high variance in the number of messages per link, since the links at the top of tree will be much more heavily used than the links at the bottom of the tree. As with the results under no contention, the network using three links per node designed by our algorithm outperforms the two dimensional torus under contention. Likewise, the network designed by GAPPIN that uses four links per node has higher performance than the three dimensional torus under contention. This shows that our algorithm can optimize networks well when including costs with and without contention at the same time.

4.3 Analysis of Modelling Contention Over Time



(a) (b)

Figure 5: Components of cost of solutions over time for 128 nodes with 8 nodes per link
(a) average latency with and without contention
(b) maximum latency with and without contention, standard deviation of # of messages per link under contention

Figure 5 shows the components of the median cost over time of the algorithm for designing a system with 128 nodes and 8 links per node, where each of the components of the cost are considered equally. It is important to note that the relationships between the components of the cost are complex. Often, to decrease the overall cost of the network, the costs of one or more components actually increase. For example, early in the algorithm, the change in cost for maximum latency under contention and standard deviation of the number of messages per link are the opposite of one another. Interestingly enough, however, the maximum latency under no contention remains constant over time, unlike the other components of the cost.

5. Maximum Wire Length Constraint

5.1 Modifications to the Algorithm for the Constraint

After looking at the networks designed by GAPPIN, as shown in section 3.1, we had some concerns about how difficult it would be to implement these networks. To solve this problem, we modified our algorithm so that it would design networks in which none of the links are longer than a maximum length. To do this, we first modified the algorithm so that the user would specify the positions of the nodes in the network. These positions may either be the absolute, physical coordinates of each of the nodes or relative positions of the nodes (for example, in a backplane system, where the nodes are in a line, node 0 would be at (0, 0), node 1 at (1, 0), node 2 at (2, 0), etc.). The user would also then specify the maximum length that a link in the network could be (in the same units as was used to specify the positions of the nodes). The algorithm estimates the lengths of links by computing the Manhattan distance from the source to the destination node. This type of estimation is often used in the placement stage of VLSI CAD tools for estimating the length of wires [21]. Manhattan distance computes the distance between (x_a, y_a) and (x_b, y_b) as:

$$\text{Manhattan Distance} = |x_a - x_b| + |y_a - y_b|$$

The next change we made to add this constraint was to add link length checking when the algorithm creates new children solutions from parent solutions (see section 2.3). When the algorithm attempts to create a link for a node in the child network using either inheritance or mutation, the algorithm must compute the estimated length of the link. If the length of the link is less than the maximum allowed length, the algorithm places the link in the child network. Otherwise, the algorithm attempts to create a new link using inheritance or mutation and keeps creating new links until it either finds a link that has a legal length or the algorithm reaches a threshold. When the algorithm reaches the threshold, it gives up on using inheritance to create the link and switches to solely using mutation. Again, the algorithm rejects links that are too long and keeps trying until it reaches another threshold. If the algorithm reaches the second threshold, it gives up and leaves the link on the node empty. Note that we also added a similar process to the generation of the initial population to ensure that the initial population (see section 2.1) of solutions also meets the length constraint.

It is interesting to note that with this simple modification, the algorithm performs a different type of network design. Without the constraint, the algorithm designs a logical topology for the network, which later must be turned into a physical. This is the same as what happens in conceptual topologies, where the topologies themselves do not dictate a particular layout. For example, there are many different ways in which one can arrange networks based on hypercubes and toruses. However, using the new length constraint, GAPPIN designs both the logical topology and physical layout at the same time, since it now considers the positions of the nodes. This is useful, as it saves designers much of the work of mapping a topology to a physical layout.

5.2 Results of Maximum Wire Length Constraint

To measure the results of our algorithm with the maximum wire length constraint, we tested the algorithm using networks in backplane and centerplane configurations. In a backplane configuration, all of the nodes in the system are arranged in a line. In contrast, centerplane configurations place the nodes in the system in two parallel lines (such that links between the nodes are put between the two lines). We chose to use backplanes, since many of the conceptual topologies with high performance (such as multidimensional hypercubes and toruses) cannot be laid out in a backplane without using one or more long links. Since centerplane networks are a simple extension of backplane networks, we decided to include these as well.

Since it is easier to visualize and analyze small systems then large systems, we started with an eight processor system in a backplane configuration with three links per node and a maximum link length of three node crossings (i.e., from node 0 to 1 is length one, 0 to 2 is length two, etc.). Figure 6a shows the physical layout of the resulting network that GAPPIN designed. To better understand this network, we rearranged it into a logical topology, which is shown in Figure 6b. Like the eight node network designed without maximum length constraints (see section 3.1), the topology of this network is similar to a three dimensional hypercube. Unlike the network without maximum length constraints, this network is not symmetric. Table 3 compares the performance of this network to other networks. Not surprisingly, GAPPIN cannot design as high of a performance network with the constraint as without the constraint. However, our algorithm still manages to build a network with higher performance than a hypercube, which cannot satisfy the length constraint in a backplane design.

To analyze the results of GAPPIN with the length constraint for larger systems, we next looked at sixteen node systems in both backplane and centerplane with four links per node and a maximum wire length of four node crossings. The sixteen node centerplane configuration is similar to that of Sun Microsystems' Starfire, which is used in Enterprise servers [22, 23]. Figure 7 shows the resulting physical layouts of these networks from GAPPIN. Note that in the sixteen node systems, we were unable to rearrange the design into a logical topology in a geometrical shape. Table 4 compares the results from GAPPIN to conceptual topologies. Like in the case of eight nodes, GAPPIN was able to design a network in a backplane configuration that has slightly lower latency than a hypercube, which cannot satisfy the constraint in a backplane network. Not surprisingly, GAPPIN designed a network for a centerplane configuration that has higher performance than that of the backplane configuration. Our algorithm is able to do this, since the centerplane effectively offers the algorithm more possible legal networks to work with and links can legally reach farther in the network. In addition, the centerplane network that GAPPIN designed has significantly lower latency than a hypercube.

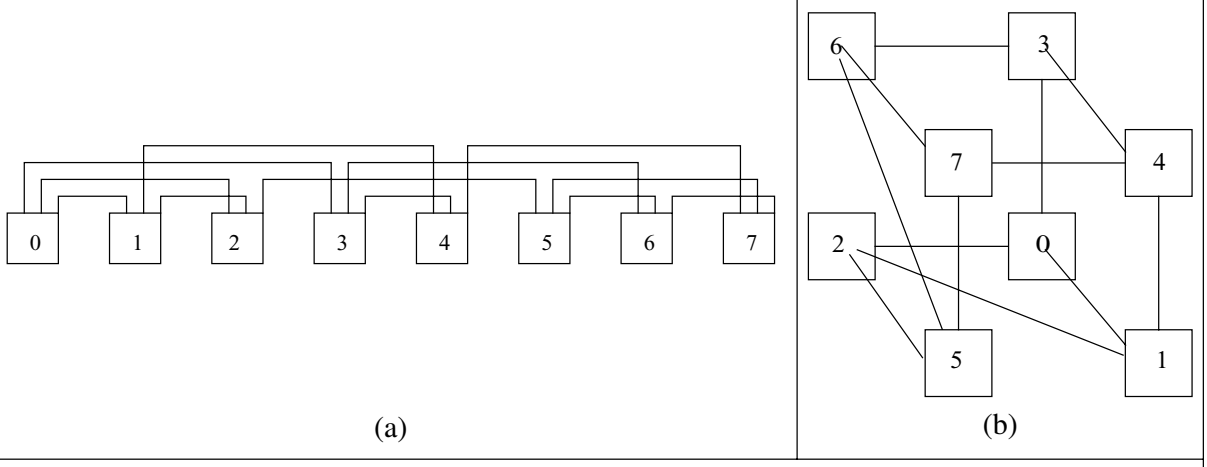


Figure 6: Graphical representation of results of network in backplane configuration with 8 nodes and maximum wire length of 3

(a) Physical layout of network

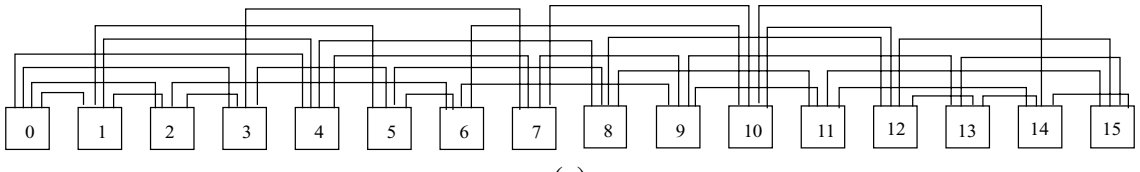
(b) Topology of network in (a)

Name	# of Nodes	Maximum Wire Length	Meets Constraint?	Average Latency	Maximum Latency
Ring	8	3	Yes	2	4
Tree	7*	3	Yes	1.95	4
3-D Hypercube	8	3	No	1.5	3
GAPPIN (constrained)	8	3	Yes	1.438	3
GAPPIN (see sect 3.1, not constrained)	8	3	No	1.375	2

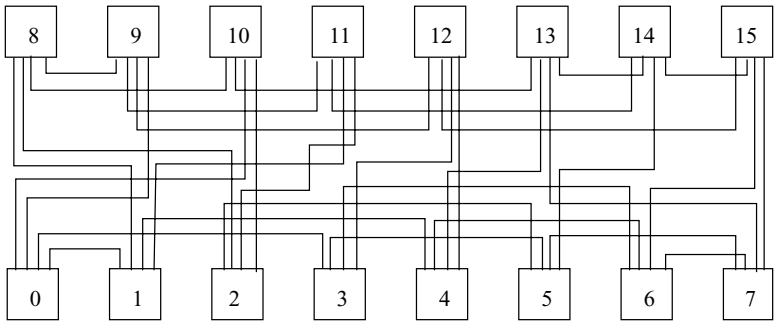
Table 3: Comparison of performance of conceptual topologies with performance of networks designed by GAPPIN for 8 node systems in a backplane configuration

* Note that the tree has only 7 nodes, since it becomes highly unbalanced with 8 nodes, making the 8 node tree not compare well with 8 node topologies

.



(a)



(b)

Figure 7: Graphical Representation for results of networks in backplane and centerplane configurations with 16 nodes and maximum wire length of 4

(a) Layout of network in backplane configuration

(b) Layout of network in centerplane configuration

Name	# of Nodes	Maximum Wire Length	Meets Constraint?	Average Latency	Maximum Latency
Ring	16	4	Yes	4	8
Tree	*15**	4	Yes	3.27	6
4-D Hypercube / 2-D Torus	16	4	backplane: no centerplane: yes	2	4
GAPPIN (backplane)	16	4	Yes	1.953	4
GAPPIN (centerplane)	16	4	Yes	1.836	3

Table 4: Comparison of performance of conceptual topologies with performance of networks designed by GAPPIN for 16 node systems in backplane and centerplane configurations

* Note that the tree has only 15 nodes, since it becomes highly unbalanced with 8 nodes, making the 16 node tree not compare well with 16 node topologies

6. Fault Tolerance

6.1 Modifications to the Network Model for Fault Tolerance

The faults that the algorithm models are node failures and link failures. A node failure occurs in the network when a node in the network stops working and it cannot send or receive any messages across the links connected to the node. This can happen, for example, if a node in a system gets a power surge, which causes the node to fail. Similarly, a link failure prevents messages from being transmitted across a link. An example of a link failure is when a cable between two nodes is accidentally disconnected. Note that we do not model message faults, which are faults where a message transmitted along the network gets corrupted. We do not model message faults, since they can be detected and corrected using ECC, parity bits. If the code can only detect an error, but not correct it, then the network can force the source node to retransmit the message. Note that the fault model that we are using is commonly used to analyze the fault tolerance of interconnection networks [25].

To route messages in the network, we assume that the network uses a dynamic routing algorithm that maintains global knowledge about the network, similar as in the case of when we modelled contention in the network. Under dynamic routing with global knowledge, when the network has no contention and no faults, messages will be routed using the shortest paths in the network. When faults occur in the network, the routing algorithm will recognize these faults due to the global knowledge and will route the messages around the faults. Therefore, the network as a whole fails when there is no possible routes are available between two or more operational nodes in the network, due to faults in the network. As such, the goal of our algorithm then is to design networks such that a dynamic routing algorithm can successfully route around multiple faults in the network with little loss in performance.

6.2 Minimum Connectivity

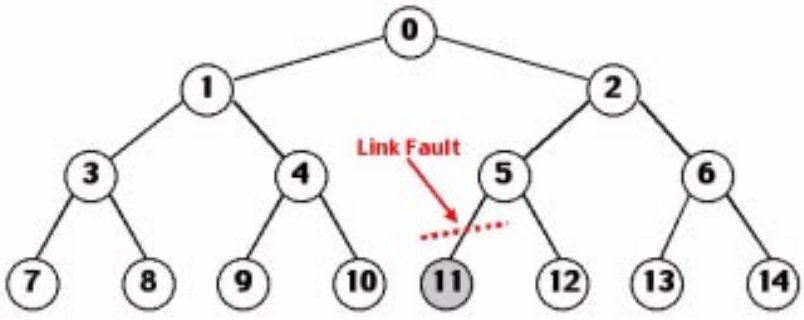


Figure 8: Minimum connectivity of a tree network, which is equal to one. The figure shows a link fault on the link between nodes 5 and 11, separating node 11 from the rest of the network.

One of the simplest cases in which the network can fail is when all of the links on a node fail, making it impossible to communicate with the disconnected node. We define minimum connectivity as the number of links connecting to the node with the smallest number of links in the network. Thus, this specifies the minimum number of node failures that need to occur to disconnect a single node from the network. Figure 8 shows an example of a tree network. Tree networks have particularly bad minimum connectivity, since only a single link needs to fail for a node to be disconnected from the network.

To compute the minimum connectivity of a network, the algorithm performs a simple linear search. In particular, the algorithm walks through each node of the network and counts how many links are on each node. The algorithm then saves the smallest number of links on a node. Therefore, computing the minimum connectivity of the network takes time $O(N \times L)$, where N is the number of nodes in the network and L is the number of links on a node. Note that this can be reduced to $O(N)$ by saving the number of links on each node when the algorithm creates each network.

6.3 Bisection Bandwidth

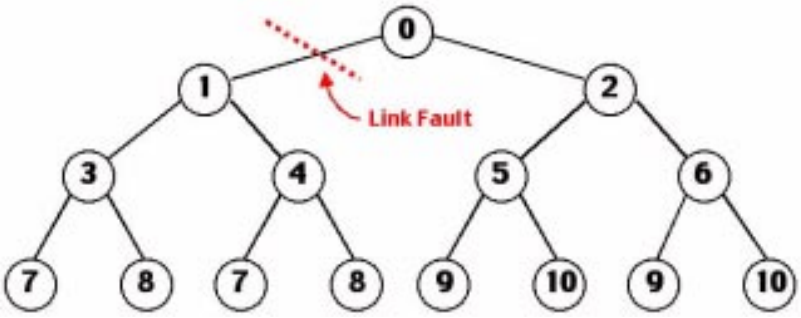


Figure 9: Bisection bandwidth of a tree network, which is equal to one. The figure shows that there is a fault on the link between nodes 0 and 1, which separates the network into two halves.

In the last section, we gave an example of a tree network and showed that it has bad minimum connectivity, which causes the network to not be able to tolerate single faults. A simple way to fix this is to mirror the leaf nodes in the network, as shown in Figure 9. Here, each copy of the leaf node performs the same steps and as long as one is connected to the network, the network still is operational. This effectively doubles the minimum connectivity and allows the tree network to tolerate single faults at the leaf nodes. However, the network as a whole still cannot tolerate single faults. In particular, if there is a link failure on the link between nodes 0 and 1 of the network, the network will be cut in half and one half of the network cannot communicate with the other half of the network.

To measure this phenomenon, we introduce a measurement called bisection bandwidth, which define as the number of links that are on the minimum cut of the network for bisecting the network roughly in half. In other words, this measures the minimum number of links that must be cut in order for the network to be cut roughly in half. Since the minimum cut may occur from cutting the network exactly in half, we instead cut the network into two partitions, each of which contains 40% to 60% of the nodes in the network. To compute the bisection bandwidth of the network, we use the Fiduccia-Mattheyses heuristic for partitioning [24]. This heuristic computes the bisection bandwidth of the network by first partitioning the network exactly in half. Next, the heuristic walks through each node in the network in turn and sees if moving the node to the other partition would decrease the number of links cut in the network. Note that a node is only allowed to move from one partition to the other if the move would maintain the balance that neither partition contains less than 40% of the nodes or more than 60% of the nodes in the network. As such, computing the approximate bisection bandwidth of the network takes $O(N \times I)$, where N is the number of nodes in the network and I is the number of iterations that we run the Fiduccia-Mattheyses heuristic (the heuristic does not always find the solution on the first iteration, so we run the heuristic multiple times to refine the result). Currently, our algorithm performs fourteen iterations of the heuristic to calculate the bisection bandwidth. Note that we need to use a heuristic

here, since computing the exact bisection bandwidth of the network is an NP-Complete problem, so we can only afford to use an approximation.

6.4 Distribution of Paths Through Links and Nodes

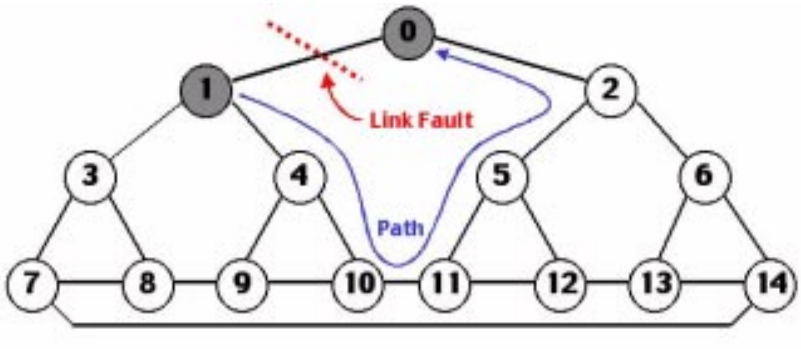


Figure 10: Distribution of the paths through the links and nodes of the network. The figure shows a link failure on the link between nodes 1 and 2, which forces messages from node 1 to 0 to take a long route.

In the last section, we gave an example of a tree network and showed that tree networks have low bisection bandwidth, which causes the network to not be able to tolerate single faults. By observing that our network model specifies that each node in the network has the same number of links, one can easily see that the leaf nodes have two empty links on each leaf node, since they only use one link each. Therefore, a simple method for increasing the bisection bandwidth of the network is to link each of the leaf nodes to its two nearest neighbors and a wrap around link from the first to the last leaf node, as shown in Figure 10. This increases the bisection bandwidth of the network from one to three. Notice that this also increases the minimum connectivity of the network from one to two, without having to create multiple copies of the leaf nodes, since the leaf nodes now use three links each. Thus, the network can now continue to operate under single faults in the network.

While the network now can tolerate single faults, its performance will degrade by a large amount under single faults. For example, if the link between nodes 0 and 1 fails, messages from node 1 to 0 must now take a much longer route than before, reducing the performance of the network. The reason for this is that there is a large variation in the distribution of messages that travel across the links of the network. In particular, the link between nodes 0 and 1 and the link between 0 and 2 will have a large amount of routes cross them, since they make routes to and from nodes at the top of the tree shorter. At the same time, links at the leaf nodes of the tree will have much fewer routes across them, since there are more routes between different leaf nodes. Since the links at the root of the network are used much more than other links in the network, there will be a large performance loss if one of these links fails. Conversely, if there is little variation in the number of messages across the links of a network, then if a link fails, there will be less performance loss, since other links will be able to handle the additional traffic with less performance

loss. Note that it is important to also look at the variation in the number of messages that pass through each node, since node failures can cause an equally bad degradation in performance. For example, in the example above, if node 0 fails, the network will perform much worse.

To compute the variation in the number of routes that use each link and node in the network, we keep track of the paths that are used when we compute the latencies of transmitting messages between nodes in the network. Note that this makes the algorithm computing latencies for transmitting messages much more complex, since the algorithm now not only needs to compute the routes between nodes by the shortest paths, but also by the least used of multiple paths of the same length. After computing the latencies transmitting messages, we collect the paths that were used and count how many times each node and link appeared in a path. Finally, our algorithm computes the standard deviation of these counts.

6.5 Single Node Failures

As we have shown in previous sections, the last three measurements can compute whether a network can tolerate single link failures and indicate what the impact will be on the performance of the network under single link failures. However, the last three measurements do not address node failures much. Therefore, we added a fourth measurement for computing the performance of the network under single node failures. In particular, we want to make sure that the network can tolerate single node failures and the performance of the network does not degrade much due to single node failures.

To compute the affect of single node failures on the network, our algorithm walks through each node in turn and injects a node failure at the node. The algorithm then recomputes the performance of the network with the node failure. The algorithm computes the average of the average latencies under the single node failures and the maximum of the maximum latencies. Note that this will catch networks that cannot tolerate single node failures, since the average and maximum latencies will be infinite for such networks.

6.6 Random Fault Injection

Lastly, we want to see how well each network can tolerate a relatively large number of faults that are distributed throughout the network. To this end, we introduce a new measurement where our algorithm measures the performance of the network where 5% of the links and 5% of the nodes are faulty. These faults are picked at random, so that they will be uniformly distributed through the network. To make sure that we hit the representative cases, our algorithm computes the performance under random fault injection twenty times. As with the case of single node failures, the algorithm computes the average of the average latencies and the maximum of the maximum latencies.

6.7 Results of Fault Tolerance

Name of Network	# of links per node	Avg Latency	Max Latency	Minimum Connectivity	Std dev # of paths through nodes	Std dev # of paths through links
Ring	2	16.00	32	2	0.00	16.00
Tree (63 nodes)	3	6.48	10	1	574.07	229.65
GAPPIN	3	4.36	9	3	823.55	364.21
2D Torus	4	4.00	8	4	0.00	2.83
GAPPIN	4	3.60	7	3	61.94	28.74
GAPPIN	5	2.95	5	5	46.50	19.21
3D Torus	6	2.97	6	6	11.97	10.34
GAPPIN	6	2.72	5	5	39.44	16.30

Table 5: Latency, connectivity, and distribution results from using the algorithm to design 64 node networks.

Name of Network	# of links per node	Avg latency Rand faults	Max latency Rand faults	Avg latency single fault	Max latency single fault	Bisection bandwidth
Ring	2	infinity	infinity	20.99	62	2
Tree (63 nodes)	3	infinity	infinity	infinity	infinity	7
GAPPIN	3	6.00	13	4.43	11	11
2D Torus	4	5.59	8	4.01	8	16
GAPPIN	4	5.24	9	3.64	8	20
GAPPIN	5	4.73	7	2.97	6	25
3D Torus	6	5.23	6	2.98	6	32
GAPPIN	6	4.91	6	2.74	5	42

Table 6: Latency, connectivity, and distribution results from using the algorithm to design 64 node networks.

Name of Network	# of links per node	Avg Latency	Max Latency	Minimum Connectivity	Std dev # of paths through nodes	Std dev # of paths through links
Ring	2	4.00	8	2	0.00	4.06
Tree (15 nodes)	3	3.27	6	1	44.04	15.75
GAPPIN	3	2.16	4	3	8.55	5.67
2D Torus	4	2.00	4	4	0.00	2.02
GAPPIN	4	4.91	6	2.74	5	42

Table 7: Latency, connectivity, and distribution results from using the algorithm to design 16 node networks.

Name of Network	# of links per node	Avg latency Rand faults	Max latency Rand faults	Avg latency single fault	Max latency single fault	Bisection bandwidth
Ring	2	infinity	infinity	4.98	14	2
Tree (15 nodes)	3	infinity	infinity	infinity	infinity	1
GAPPIN	3	3.16	6	2.23	5	5
2D Torus	4	3.15	4	1.99	4	8
GAPPIN	4	3.28	5	2.06	5	8

Table 8: Latency, connectivity, and distribution results from using the algorithm to design 16 node networks.

Table 5 and Table 6 show the results of having the algorithm design networks for 64 node networks compared to the results from conceptual topologies. First we see that GAPPIN can design networks with lower average and maximum latencies under no faults than conceptual topologies, which matches the results where we only measured latency. The results of minimum connectivity are mixed, as our algorithm can design networks with larger minimum connectivity that tree networks, which have low minimum connectivity, but lower minimum connectivity than that of torus networks, which have high minimum connectivity. Our algorithm also has a larger variation in the number of messages that pass through nodes and links than conceptual networks. Our algorithm can design networks with lower average latency in the presence of faults than conceptual topologies. Our algorithm also has mixed results for maximum latencies under faults, as sometimes it designs networks with lower maximum latency and sometimes with higher maximum latency. Finally, our algorithm designs networks with larger bisection bandwidth than con-

ceptual topologies. Overall, we can see that our algorithm makes different tradeoffs in designing networks than conceptual topologies. Note that by changing the relative importance of the terms of the cost function will most likely allow the algorithm to make the same type of tradeoffs as conceptual topologies. In addition, our algorithm is able to design networks with three links per node that can tolerate single faults, unlike tree networks.

Table 7 and Table 8 show the results of our algorithm for designing sixteen node networks. The results for the case of three links per node are similar to the case of 64 node networks, except that our algorithm was able to design a network with lower variation in the number of links through nodes and links. However, our algorithm was not able to design a network that was better than a two dimensional network for a sixteen node network. This may be due to not performing enough generations of the algorithm and not having a large enough population size.

7. Concluding Remarks

In this paper, we have introduced a genetic algorithm for designing parallel processor interconnection networks. This algorithm can design networks based on average and maximum latencies. We have shown that this algorithm can produce better results than conceptual topologies, which is the current approach for designing such networks. In particular, we found that GAPPIN can create a network using three links per node with lower latencies than a two dimensional torus, using four links per node. In addition, we found that GAPPIN can design a network using four links per node that has lower latencies than a three dimensional torus, which uses six links per node.

We also showed that our algorithm can design networks under constraints which make it difficult to design using conceptual topologies. In particular, we added a maximum wire length constraint to our algorithm, which many conceptual topologies could not satisfy in backplane configuration. However, our algorithm was able to design networks for these configurations and still manage to come up with high performance designs.

We introduced five cost measurements for measuring how fault tolerant a network is. These measurements are: minimum connectivity, bisection bandwidth, variation in the number of paths through nodes and links, performance under single node failures, and performance under multiple random faults. As we have shown, these measurements are effective for understanding how fault tolerant a network is. There is still some work that can be done to study measurements for fault tolerance. For example, one should study more what the fault coverage is for the measurements we included and see if there are other measurements that can be included to increase the coverage. An example of a cost measurement that we did not have time to implement is bisection node bandwidth, which is the minimum number of nodes that need to fail to cut the network roughly in half (note that this is a more difficult heuristic to implement than bisection bandwidth, since one would need to deal with Steiner trees instead of point to point graphs). Another improvement to these measurements would be to improve the heuristics used to compute these measurements, since some of these measurements have significant error, especially when working with large networks.

To optimize networks for fault tolerance, we included five cost measurements. We showed that with the addition of these cost measurements, the algorithm can be quite effective at designing fault tolerant networks. As we have seen, our algorithm can design networks with better fault

tolerant measurements than conceptual topologies, such as in the case of latencies of the network with and without faults. However, the algorithm often cannot design networks with better measurements than conceptual topologies, such as in the case of variation in the number of paths through node and links. Currently, it is not clear whether the algorithm poorly designs networks for these measurements due to something inherent in the manner that it designs networks or if this occurred because of the weights of the terms in the cost function that we used.

Another benefit of the algorithm is that it provides an automated method for creating interconnection networks. We envision that as the algorithm matures, engineers will use the algorithm to completely design the network and then build the network in the manner that the algorithm specifies. This will help reduce the length of time that it takes to design an interconnection network.

In general, we found that it is relatively simple to modify our algorithm to include whatever design constraints or optimize for whatever performance metrics one wants. These constraints and optimizations do not fundamentally change the algorithm itself. For example, one modification that we have added and are currently studying is including latencies of the network under moderate and high contention, using a dynamic routing algorithm. We believe after one designs a network using the algorithm and then designs the routing algorithm for the network, it may be useful to feed the routing algorithm back into GAPPIN to see if the new network it designs is different from the original network. Another modification to our algorithm that we are working on is minimizing the total length of the wires in the network, so as to reduce the cost of the network.

We have many different plans for improving our algorithm. First, we would like to study more about what constraints would make it easier to implement the designs made by GAPPIN. For example, many designs have constraints as to how many layers of links may exist and cross over each other. One possible method is to modify the algorithms that CAD designers use for routing VLSI chips [13-19]. Second, we would like to study the cost function more to better understand exactly what are good measurements for the cost of the network and how to measure these costs. Likewise, we would like to better understand the interactions between the components of the cost function and the result of the relative weighting of the components of the cost. Lastly, we would like GAPPIN to be able to detect and remove isomorphic networks in a generation. Isomorphic networks are networks that are equivalent and can be made the same by renaming the nodes in the network. By removing isomorphic networks, GAPPIN would converge more quickly to a final solution, since it would effectively reduce the size of the solution space.

References

[1] T. Lau and E. Tsang. Applying a Mutation-Based Genetic Algorithm to Processor Configuration Problems. In *Proceedings of ICTAI-96*, August 1996.

[2] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of ASPLOS VII*, October 1996.

[3] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, July 1997.

[4] D. Burger, et al. DataScalar Architectures. In *Proceedings of the 24th International Symposium on Computer Architecture*, July 1997.

[5] C. Leiserson, et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the ACM on Parallel Algorithms and Architectures*, June 1992.
[6] S. Scott, et al. Performance of the SCI Ring. In *Proceedings of the 19th International Symposium on Computer Architecture*, July 1992.
[7] J. Emer and N. Gloy. A Language for Describing Predictors and its Application to Automatic Synthesis. In *Proceedings of the 24th International Symposium on Computer Architecture*, July 1997.
[8] R. Friedburg. A Learning Machine: Part I. *IBM Journal*, 1958.
[9] R. Friedburg, et al. A Learning Machine: Part II. *IBM Journal of Research and Development*, 1959.
[10] M. Benton and S. Sait. A Genetic Algorithm Approach to Optimize Two-Bit Decoder PLAs. *International Journal of Electronics*, January 1994.
[11] J. Cahoon and W. Paris. Genetic Placement. *IEEE Transactions on Computer Aided Design*, November 1987.
[12] K. Shahookar and P. Mazumder. VLSI Cell Placement Techniques. *ACM Computing Surveys*, June 1991.
[13] S. Akers. A Modification of Lee's Path Connection Algorithm. *IEEE Transactions on Electronic Computers*, February 1967.
[14] F. Hadlock. A Shortest Path Algorithm for Grid Graphs. *Networks*, 1977.
[15] D. Hightower. A Solution to Line-Routing Problem on the Continuous Plane. In *Proceedings of the 6th Design Automation Workshop*, 1969.
[16] C. Lee. An Algorithm for Path Connection and its Application. *IRE Transactions on Electronic Computers*, 1961.
[17] K. Mikami and K. Tabuchi. A Computer Program for Optimal Routing of Printed Circuit Connections. In *Proceedings of IFIP*, 1968.
[18] J. Soukup. Fast Maze Router. In *Proceedings of the 15th Design Automation Conference*, 1978.
[19] J. Soukup and J. Fournier. Pattern Router. In *Proceedings of ISCAS*, 1979.
[20] T. Cormen et al. *Introduction to Algorithms*. Cambridge: The MIT Press, 1990.
[21] S. Sait and H. Youssef. *VLSI Physical Design Automation*. London: McGraw-Hill International, 1995.
[22] *Sun Enterprise 10000 Server: Technical White Paper.* Sun Microsystems, 1998.
[23] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, January-Februrary 1998.
[24] C. Fiduccia and R. Mattheyses. "A Linear Time Heuristic for Improving Network Partitions." In *Proceedings of the 19th Annual Design Automation Conference (DAC-19)*, June 1982.
[25] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Third Edition Beta. San Francisco: Morgan Kaufmann Publishers, Inc., not yet published.
[26] M. Thottethodi, A. Lebeck, and S. Mukherjee. "Self-Tuned Conjestion Control for Multiprocessor Networks." In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA-VII)*, January 2001.