instance, metal wires on VLSI chips may be a few millimeters long and spaced a few microns apart, whereas wires connecting different boards may be as long as a 1 meter, and spacing between wires on a cable connector may be as much as 1 mm. This results in a higher cost per wire at the higher levels of the hierarchy, and the higher cost translates into a reduction in number of wires available at the higher levels. *Packaging locality* is also seen when we consider memory hierarchies [ACF90], where the bandwidth at lower levels (e.g. registers) is higher than the bandwidth at the upper levels (e.g. DRAM). We believe that *packaging locality* is a fundamental property that will be applicable to all future packaging technologies as well.

A consequence of *packaging locality* is that the costs at the higher levels of the packaging hierarchy dominate the total cost of the network. Accordingly, we define three generic models that emphasize the costs at the higher levels of the hierarchy. These models are briefly described in Section 3.1. The three models progressively characterize costs at increasing levels of detail. Each model is the topic of a subsequent chapter, where we evaluate different network designs under that model.

Section 3.2 outlines our network design strategy. We design networks in a hierarchical fashion starting at the top, proceeding down the levels of the hierarchy. In Section 3.3 we describe the work-load model and the types of messages we expect the network to handle. We approximate this work-load using synthetic work-loads and evaluate the networks by simulating these work-loads. Our synthetic work-loads assume that the messages in the network are sent to randomly chosen destinations. Section 3.3.1, describes our network evaluation methodology. Sections 3.3.2 and 3.3.3 describe the synthetic work-loads in detail. Finally in Section 3.4, we provide details regarding the routing algorithm, messages queueing mechanism, message lengths, etc.

3.1 Generic Models

We start with a very simple model and gradually move on to more complicated models. Since we expect the costs at the higher levels of the packaging hierarchy to dominate the cost of the network, for our first generic model, *model 1*, we shall limit ourselves to the costs at the top level of the hierarchy. Note that the actual hierarchy may consist of several levels of packaging, but since *model 1* only deals with the costs at the top, it is sufficient to consider a two level hierarchy: level 0 and level 1. Everything below the top level of the hierarchy is modeled as being internal to a level-1 module, and the costs internal to a level-1 module are ignored. The parameters describing the interconnection between the level-1 modules are used to compute cost. In our model described in Section 2.1, we used four parameters to characterize the cost at each level: the number of modules, the pin-count, the number of bundles and the bisection width. To keep *model 1* simple, we model the cost of the network using only two of these four parameters: the number of level-1 modules, and the level-1 pin-count. Although under *model 1*, the level-1 bisection width does not contribute to the cost of the network, we shall see that when two networks have different bisection widths, it is possible to show that the one with the lower bisection width will have lower performance. Therefore, we shall see that for the networks we compare under *model 1* (Chapter 4), the level-1 bisection width is held constant. However, *model 1*, ignores any restrictions on the number of level-1 bundles. The costs at level-0 are also ignored.

In order to fully define *model 1*, we also need to define the cost function. Since we have decided to compute the cost of the network using the number of level-1 modules, and the level-1 pin-count, we need to specify the relative weights of these two parameters to define the cost function.

However, these relative weights are likely to be specific to a particular technology, and the weights are likely to change significantly when the underlying technology changes. Therefore, any results obtained under a particular choice of weights are likely to be applicable only to the specific technology that the weights correspond to. Since we are interested in obtaining results that are applicable under a wide range of packaging technologies, we seek a technology independent way of computing the network cost.

The approach we take in this thesis is to retain the cost of a network as a multi-dimensional quantity. Cost, then becomes a vector rather than a simple number. Under *model 1*, the two components of the cost vector are the number of modules and the pin-count. When cost is a multi-dimensional quantity, it becomes more difficult to compare the costs of networks. If we have networks with different costs, we can only specify a partial order on the costs of the networks, because the cost of one network is greater than the cost of another only if all the components of the cost vector of the first network are greater than the corresponding components of the second. When we only have a partial order, the set of cost comparisons we can make is limited. However, we shall see that even with just a partial order available to us, we can still draw qualitative conclusions regarding the design of networks.

The second generic model, *model 2* is more detailed than *model 1*; but it is also based on a two level hierarchy. In going from *model 1* to *model 2* we add one more dimension to the cost vector; namely the number of level-1 bundles. Therefore, the cost vector has three dimensions: the number of level-1 modules, the level-1 pin-count, and the number of level-1 bundles. As in *model 1*, we shall see that the bisection width of the network is held constant, even though it does not form part of the cost vector. Therefore, *model 2* takes into account all the parameters of the top level of the packaging hierarchy. Similar to *model 1*, we ignore the costs at level-0. We shall see that when we use a more complex cost vector, the search space of interesting network designs widens. For example, the each of the dimensions of the cost vector of one network might have been lower the corresponding dimensions of another under the simpler cost vector of *model 1*, but when we make the cost vector more complex, we may not have a similar domination in terms of the dimensions of the new cost vector. We shall see that comparing the costs of networks under *model 2* is more difficult than under *model 1*, for this reason. Detailed descriptions of network evaluations under *model 2* are presented in Chapter 5.

In *model 3*, we increase the level of detail in the modeling of costs by extending the number of levels in the hierarchy. In *model 3*, we consider the costs at the top two levels of the hierarchy. Therefore, it is sufficient to think of the hierarchy as consisting of 3 levels. The cost vector under *model 3* is more complex than the cost vectors of the other two models. In *model 3* the cost vector has five components: the number of level-2 modules, the number of level-1 modules, the level-2 pin-count, the number of level-2 bundles and the level-1 pin-count. The top-level bisection width (level-2 bisection width), is held constant as in the other two models. Since we have a large number of dimensions in the cost vector, we shall fix some of the dimensions and explore the trade-offs between the others. Specifically we shall fix the number of top level modules, the number of level-1 modules and the level-2 pin-count and explore the trade-offs between the level-1 pin-count and the number of bundles. Details of the networks that exhibit these trade-offs are presented in Chapter 6.

3.2 Design Strategy

We wish to design the best network under each of the three models described above. We pose this design problem as a search problem, where we compare the performance different networks of fixed cost. A network is defined by specifying the interconnections at each level of the hierarchy. There are several choices at each level and we are confronted with the cross-product of the design choices. Evaluating each of the designs in this cross-product can be time consuming because the set of possible designs can be very large. Therefore, we seek a better strategy of exploring this search space.

We adopt a hierarchical design strategy where we start at the top and go down the levels of the hierarchy sequentially. For instance, in the case of our first generic model, *model 1*, we design the level-1 network first and then consider the level-0 network. When we design the top level network, we ignore the costs at level-0, and design it so as to maximize the performance for a fixed cost. Then we proceed to level-0 and consider the design of the network at that level. Let us assume that the level-0 network costs a fraction x of the cost of the top level network, for a total cost of $(1 + x)$. It is conceivable that we might have been able to design a better network for the same cost of $(1 + x)$, had we considered the cross-product of possible designs at both levels. As we shall see later, the design of the level-1 network has a greater impact on overall performance than the level-0 network. Therefore, a network that is better than the one our strategy yields, is expected to have a higher cost allocated to the top level network. However, we expect that the fraction x is likely to be small to begin with, since we believe that for most technologies, the costs at the higher levels tend to dominate the total network cost. Besides, we also expect that the amount of cost reallocation possible is also likely to be small. Therefore, we believe that our strategy is justified.

While designing the top level network we do not have a design for the lower level network, so we can only estimate the performance of the top level network, instead of measuring it. We estimate the performance of the network using its *peak throughput*. The *peak throughput* of a network is the maximum amount of data that it can deliver in steady state, in unit time when the communication traffic is randomly distributed. We compare several alternatives for the top level network and choose those that have the highest peak throughput for the least cost.

We then proceed to the next lower level of the hierarchy and consider design alternatives at that level. We prune the number of design choices using a strategy similar to the one applied to the top level network, namely choosing networks that offer high estimated performance for low cost. Since the top-level network fixes the maximum performance that can be achieved, when we design the lower level networks, our objective is to maximize the utilization of the top-level network, and we attempt to achieve a level of performance that is as close as possible to the peak. After we have designed a network for each level, we evaluate the entire network using detailed simulation.

The level-by-level design strategy simplifies the network design process, and is justified from the point of view of packaging costs, but we may potentially run the risk of eliminating good networks from consideration apriori. For instance, we eliminate a top-level network that has a lower peak performance. Consider a situation where we have a choice between two top-level networks A and B. Let the peak performance of B be 0.8 that of A. Further let there exist lower level networks for B that achieve 0.8 of its peak performance. Unless we can design lower level networks for A that achieve greater than 0.64 of its peak, we would have skipped over a better network design since we would have eliminated B because it had a lower peak performance. However, we shall see that for most of the networks we evaluate, we are able to design lower level networks that achieve close to the peak performance of the top-level network reducing the risk of eliminating better networks without evaluating them.

3.3 Work-load Model

All of our network evaluations are based on randomly distributed inter-processor communication traffic. While some applications, such as those based on dense matrix computations, have extremely regular communication traffic, there is a large class of problems for which the communication traffic tends to be highly irregular and varying with time. Sparse matrix computations and graph algorithms belong to this latter class. Further, many programs that initially had regular communication patterns tend to lose this property when algorithmic optimizations are performed. For instance, the $O(n^2)$ version of the N-body program can be written so that processor i only has to communicate with processors $(i+1)$ and $(i-1)$. However, when a better algorithm such as the $O(n \log n)$ Barnes and Hut algorithm [BH86] or the $O(n)$ Greengard and Rokhlin algorithm [GR87] is used, the communication traffic ceases to be as regular as in the $O(n^2)$ case. Further, for many programs, the regularity in communication patterns is lost when the size of the problem is much larger than the number of processors available. For instance, an FFT computation where the number of processors is equal to the number of elements in the FFT, has a communication pattern that maps well onto a hypercube, but if the number of elements in the FFT is substantially greater than the number of processors, an efficient implementation generally requires communication traffic that is no longer hypercube-like. For all of the above reasons, it is essential to design networks that are capable of supporting global communication traffic efficiently. In this thesis we model global communication using randomly distributed communication requests (the communication requests issued by each processor are assumed to be distributed uniformly across the entire machine).

For simplicity, we assume that the communication primitives are based on a shared-memory model, but we believe that our results do not rely heavily on this assumption. The logical view of the machine is shown in figure 3.1. The machine consists of N_p processors connected to N_m memory units using an interconnection network. The processors communicate with each other by reading or writing locations in the shared address space. The shared address space is distributed among the memory units. When a processor needs to access a location in the shared address space, it sends a message to the corresponding memory unit. The message travels from the processor the memory unit, performs the access and returns to the processor. Each access corresponds to a short message that typically deals with a small amount of data, such as a single word or a cache line. In addition to the memory units shown in the figure, the processors may have private local memory and caches which are not shown.

The logical view of the machine encompasses a variety of different architectures. It includes machines like the IBM RP3 where all the memories are equidistant from all the processors, and machines where the latency associated with different accesses are different because the path lengths to the corresponding memory units are different. It also includes bus-based shared-memory machines with a single memory unit sitting on a bus ($N_m = 1$) and machines which consist of processor memory pairs. In the latter case, $N_p = N_m$ and each processor may have a single memory system which is partitioned into private and shared memory. For most of the machine architectures evaluated in this dissertation N_p will be equal to N_m, but we also consider a few cases where $N_m < N_p$.

We assume that each time a processor wishes to make an access to shared-memory, it directs the access to a randomly chosen memory address. Access non-uniformities and *hot-spots* [PN85] due to repeated accesses to synchronization variables were topics of active research in the recent past. Message combining was proposed as a means of dealing with hot-spots and was implemented in a few research machines [GGK$^+$83, PBG$^+$85]. See [LKK86] for an evaluation of message combining techniques. However, combining-networks tend to

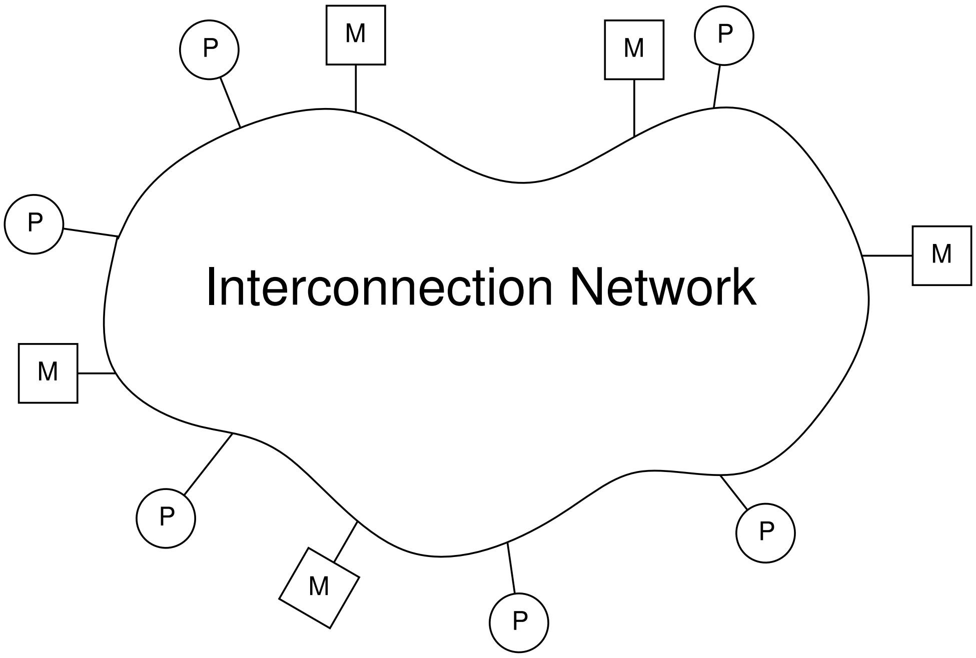

Figure 3.1: Logical machine model

be expensive to build and are sometimes not justified since software techniques such as the ones proposed in [YTL86] and [MCS91] can be used to prevent hot-spots. Nevertheless, combining-networks may be justified for certain specialized functions such as barrier synchronization [L^+92]. If the hot-spot accesses are handled either using software techniques or by the use of special purpose networks, the remaining accesses will tend to be uniformly distributed. Therefore, in the remaining part of this thesis we will not deal with hot-spots explicitly.

3.3.1 Evaluation Methodology

We use simulation of synthetic work-loads to evaluate network performance. We prefer simulation to other options such as prototyping and analytical modeling for a variety of reasons. Prototyping is both expensive and time consuming. It is also unfeasible when a large number of networks need to be evaluated. Solving analytical models can be intractable unless simplifying assumptions are made, and it may not always be possible to make such assumptions without sacrificing the accuracy of the results. Under these conditions, the only available option is simulation. Simulation models can be much closer to the actual network since the only real limitation of model complexity is simulation time.

We use synthetic work-loads to drive the simulation instead of real parallel programs or execution traces. We evaluate systems consisting of a thousand processors, and since only a small number of such systems have been built, it is difficult to get a good collection of real programs. Besides, many parallel programs that have been written, tend to be proprietary and therefore unavailable. It might be possible to get traces of these proprietary programs, but it is not clear whether a trace obtained on one machine can be used to evaluate another with a different network architecture and timing. Even if we were able to configure the trace to take differences in network architecture and timing into account, simulation of the trace will only tell us how the architecture performs on that particular program. By using a synthetic work-load and varying its parameters, we can exercise the network under a wide range of conditions leading to a better evaluation of network performance.

3.3.2 Open-Network Model

The first work-load model we use is the well-known *Open-Network* [Lee89] model and has been used by many researchers. Under this model, each processor makes an access in each cycle with probability λ and performs a local operation with probability $1-\lambda$. Each access is directed to a random address that is chosen uniformly over the address space. The activity of any two different processors are independent of one another. Processors continue execution without waiting for their previous accesses to complete; hence the name *open-network* model.

The interval between successive accesses made by a processor is a geometrically distributed random variable with a mean of $1/\lambda$. The accesses form a discrete time version of a *poisson process* that is commonly used in queueing theory. The message generation process is memory-less, i.e., the probability of generating a message in a given cycle is independent of the history of the system.

We simulate each network under the open-network work-load for different values of λ and measure the average latency at steady state. Results are reported as graphs of latency versus the offered load λ. We compute the average latency at 1000 cycle intervals and when the last 10 observations (corresponding to 10,000 cycles) lie within 2% of each other, we conclude that the network has reached steady state and record the latency. If the offered load λ is greater than the bandwidth that the network can provide, the latency does not stabilize and therefore there is no data point for the corresponding value of λ. The interesting metrics under this work-load model are the maximum sustainable value of λ and latency.

3.3.3 Multithreading Model

In the open-network model, network latency has no influence on message generation. It is unreasonable to expect that a processor can have an unbounded number of outstanding messages waiting in the network. Therefore, it is desirable to incorporate feedback in the message generation process to make the work-load model more realistic.

Incorporating this feedback using an execution model based on a simple uniprocessor can lead to poor performance. In the case of a simple uniprocessor, the processor has to wait for each memory access to complete before it can continue executing. This is inappropriate for large scale parallel machines because each access can typically take a large number of cycles to complete. Our measurements indicate that it is reasonable to expect access latencies of approximately 100 processor cycles for a 1000 processor machine. If a processor has to wait for each access, the processors will spend most of their time idling. For example if the memory access takes 100 cycles and the processors make a shared access every 20 cycles, processor utilization can never be more than 16%. Further the network will be significantly under-utilized because each processor can have only one access in the network at any given time.

It is essential for the processors used in large scale parallel machines to have a mechanism to tolerate long access latencies. We shall first assume that remote accesses are split-phase operations. Each remote access consists of two instructions: an *initiate* instruction and a *use* instruction. When a processor needs to make an access, it executes the *initiate* instruction which creates a message and sends it over the network. Later, the processor executes a *use* instruction, which causes the processor to stall for the corresponding access to complete. We assume that both *read* and *write* accesses are acknowledged. In response to a *use* instruction, the processor waits for the corresponding acknowledgement to be received. By breaking up a remote memory access into *initiate* and *use* instructions, we may be able to overlap the memory access latency with useful computation. Typically a compiler would first split a remote memory access into *initiate* and *use* instructions and then attempt to move the *initiate* instructions backward so that the memory access is initiated many cycles before the result is required. It may also be possible to pipeline multiple memory accesses by executing successive *initiate* instructions before the corresponding *use* instructions.

In addition to split-phase accesses, we assume that each processor has the capability to run multiple threads of execution and the ability to switch between execution contexts rapidly [Kow85, A+90]. If the processor has multiple register sets, (one per thread) and we use a simple round-robin thread scheduling mechanism, it is possible to context-switch in a single processor cycle [Boo93]. We assume that the processor has a *context-switch* instruction which causes the processor to suspend the current thread and start executing the next thread (in round-robin order). The task of the compiler is to insert *context-switch instructions* before any *use* instruction that can be expected to stall the processor. By using such a multithreading mechanism, the processors can tolerate much longer latencies than possible by merely using split-phase instructions. It is also possible for each processor to have multiple outstanding accesses in the network, thereby running the network at reasonable load.

The multithreading work-load model is defined based on the above view of processor execution where each processor runs multiple threads that perform split-phase accesses. Each processor is assumed to be running t threads using a round-robin thread scheduling policy. When a thread starts executing, it first executes a *use* instruction which stalls the processor till its previous accesses to completes. In the next cycle, the thread initiates a remote access. Then the thread executes a sequence of *local* instructions followed by a *context-switch* instruction. The *context-switch* instruction suspends the thread until the processor has stepped through the other threads. Each thread repeatedly executes the following sequence of instructions: Initiate, local, local, local, ..., Context-switch, Use. The accesses are tagged so that each *use* instruction waits for the access issued by the corresponding *initiate* instruction.

We can control the interval between accesses by varying the number of local instructions executed between the *initiate* and *context-switch* instructions. We use two inter-access interval distributions: geometric and periodic. In the geometric model, in each cycle following the *initiate* instruction, the processor either executes a *context-switch* instruction with probability p or a *local* instruction with probability $1 - p$. In the periodic model the number of local instructions is constant. If the processor is executing a tight inner loop, the interval between accesses is likely to be constant. The periodic model attempts to capture this effect. In reality the distribution of access intervals is likely to be somewhere in between the two models.

In both the periodic and the geometric models, the accesses generated by a processor were assumed to be distributed randomly across the entire memory address space, as in the previous open-network work-load model. Note that even though the access intervals are constant in the periodic model, the accesses typically take different paths in the network depending on their destinations, and contention for message paths leads to queueing of messages at routing nodes.

We simulate the networks under both the periodic and geometric models for different numbers of threads per processor and for different values of the mean interval between accesses. The performance metric of interest, is the average processor utilization rather than the latency or the bandwidth of the network. As in the previous case, we measure average processor utilization at 1000 cycle intervals and when 10 consecutive observations (corresponding to 10,000 cycles) fall within 2% of each other, we conclude that the network has reached steady state and record the value. We note that we will always have a steady state value for the processor utilization since the network can never become unstable. Because of the feedback in the message generation process, an increase in network latency results in a decrease in processor utilization which results in a decrease in the network load. Eventually the processor utilization stabilizes, at which point, we record its value.

3.4 Routing Algorithm

We assume that all networks use virtual cut-through routing [KK79] to forward messages. Cut-through routing achieves high bandwidth that is a characteristic of packet switched routing while simultaneously providing low latency similar to circuit switched routing. In cut-through routing, each message is divided into multiple flits that are sent in a pipelined fashion, one behind the other. The head flit of a message contains the routing information needed by a node to decide where the message is to be sent. When a routing node receives the head flit of a message, it immediately attempts to forward the head flit without waiting for the entire message to come in. If a message cannot be forwarded because the output channel requested by the message is busy, or there are other messages queued in front of it, the node buffers the flits of the message.

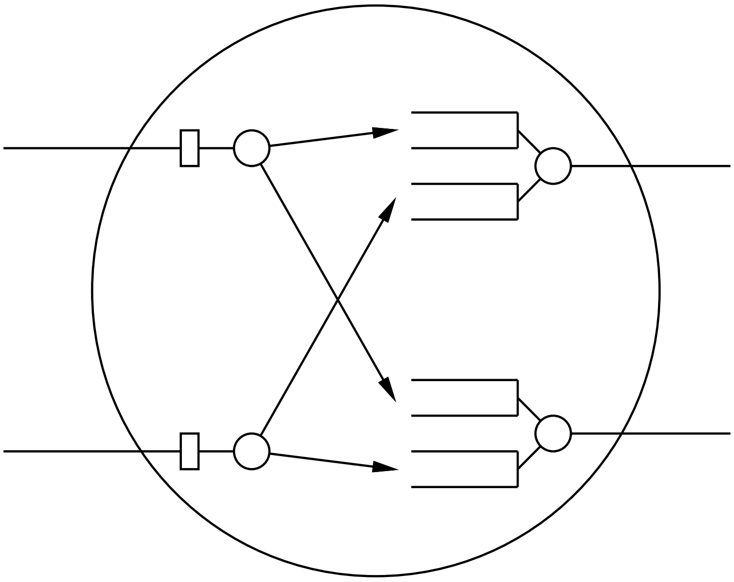

Figure 3.2: Buffers in a routing node

Virtual cut-through routing is different from worm-hole routing in that when the head flit of a message is blocked at a node, that node accepts the subsequent flits of the message and buffers the flits. In worm-hole routing a blocked message occupies buffer space on various nodes along its path. The primary difference between worm-hole routing and cut-through routing is the amount of buffer space available at the routing nodes. In cut-through routing each node is required to have sufficient buffer space to buffer entire messages and when a message is accepted for routing, buffer space is allocated for the entire message.

The allocation of buffer space does not pose a significant problem since our networks need to handle only short messages. Since we assume a shared memory model, messages sent over the network are typically accesses to shared memory — single or double word fetches or stores. All messages are round-trip messages, meaning that all write accesses are acknowledged. For the sake of simplicity, we assume that both access requests and replies are 128 bits long. This message length includes data, memory/network addressing information, and control information.

The queuing mechanism within the nodes is as follows: Each node that has I inputs and O outputs, has O sets of I message queues. The I queues of each set are connected to an output link using a I to 1 multiplexer. Any message arriving on input link i which needs to go out along output link o, enters the ith queue of the oth set of queues.

Figure 3.2 shows the buffers in a 2 input, 2 output routing node. Each output link services its queues in a simple round-robin fashion. A single flit buffer is assumed to be present at each input port to simplify the problem of allocating queue space to the messages. If the message whose head flit is in the input buffer cannot be accepted because of unavailability of queue space, it is forced to wait there until space becomes available. If this happens, the corresponding link remains idle until queue space is available and the message is accepted. In all the simulations, each message queue was assumed to be capable of holding 8 messages. Therefore, a 2 input, 2 output node has 4 queues, each of size 8 messages for a total of 512 bytes of buffer memory.

We also assume that all the wires in the network can transfer one bit per clock cycle and that each routing node imposes a single cycle pipeline delay on any message that does not suffer link or queue space contention. Recall that we model differences in clock rates using increased wiring densities at the lower levels of the packaging hierarchy.

In many networks evaluated in this dissertation, the channel widths at different parts of the network are different. However, all input channels (respectively output channels) of a routing node have the same channel width. If the output channels are wider than the input channels, the routing node cannot forward the flits as they come in. We assume that these nodes use store-and-forward routing, i.e., they wait for the entire message to come in before sending out anything. For example, consider a routing node that has 16 bit wide input channels and 32 bit wide output channels. If the first flit of a message comes in at cycle t, the node waits until cycle $t + 7$ to get the entire 128 bit message. Output transmission begins at cycle $t + 8$ and the last flit of the message leaves the node at cycle $t + 11$. Waiting for the entire message is a bit conservative, because it would be possible to start transmitting the message in cycle $t + 5$ and the last incoming flit would be able to leave the node in cycle $t + 8$. Hence our assumptions are pessimistic for networks that have non-uniform channel widths.

Chapter 4

Model 1: Pin-counts

In this chapter, we consider the first of our generic models, as defined in Section 3.1. Recall that this model consisted of a two-level packaging hierarchy where the machine is partitioned into some number of level-1 modules at the top-level. In this chapter, we shall simply refer to the level-1 modules as *packaging modules* or just *modules*. There are two levels of wiring: connections between modules (inter-module) and connections within a module (intra-module). Cost is a two dimensional vector, with the number of modules and the pin-count being the two components.

To avoid degenerate designs where we put all the processors in one module, and essentially have a pin-count of zero, we shall assume that there is a limit on the maximum number of processors that we can put in a module.

4.1 Design Strategy

We would like to design the *best* network under this model. Recall that we defined the *best* network to be one that achieves the maximum performance for a fixed cost, or equivalently one that achieves a fixed performance at a minimum cost. The strategy we adopt in this chapter is to fix the cost and compare the performance of networks.

Since the cost is a two dimensional quantity, we fix the cost by fixing both the dimensions. We fix the number of modules at M and the pin-count per module at P. Also let N be the total number of processors in the machine and s be the maximum number of processors per module.

We have some flexibility in choosing the value of M. The minimum value for M is

N/s, since each module cannot contain more than s processors. We are also likely to have some maximum limit on M; if we have a large value of M, we may have to organize these M modules themselves in a hierarchical fashion, but we assumed the modules formed the top level of the hierarchy. If we choose $M = N/s$, we minimize one of the dimensions of the cost vector, and we shall choose M thus. We note that other choices are equally valid, but they represent different points in the cost space. By choosing $M = N/s$, we are restricting our attention to a smaller portion of the cost space. We believe that this portion of the cost space is interesting because this portion of the cost space corresponds to minimizing one of the dimensions of the cost vector, and also because it corresponds to partitioning the machine into symmetric modules, with each module consisting of an equal fraction of the machine. A machine in which all modules are identical, is easier to build than a machine which uses different types of modules.

We also have some flexibility in choosing a value for P. Technology may dictate an upper bound on the value for P, but any choice of P that is less than the upper bound is a valid one. As we shall see later, the performance of any network will be directly proportional to P, so we shall defer the selection of a value for P until later.

As mentioned in Section 3.2, we shall adopt a hierarchical design strategy; first we design the top-level network (the inter-module network) and then consider the lower level network (the intra-module network). For the top level design we shall estimate the performance of the network using its peak throughput for randomly distributed communication traffic. Recall that the peak throughput of a network is defined as the maximum amount of data that a network can deliver in steady state in unit time. The design objective for the top-level network is finding a network of M modules and P pins per module, that has the maximum peak throughput.

After we select the network for the top level we shall consider several alternatives for the lower level network and simulate the entire (top level + lower level network) to determine the fraction of the peak throughput that is achieved by the network.

4.2 Inter-Module network

For randomly distributed communication traffic, the bisection width of a network is a good estimator of its peak throughput. Under random traffic, half of the messages sent by each processor cross the bisection of the network. The amount of time required to deliver

Inter-Module Topology	Pins per Module	Bisection Width
Complete graph	P	$(M^2/4)\frac{P}{M-1} = O(PM)$
Butterfly	P	$(M/\log M)\frac{P}{4} = O(PM/\log M)$
CCC	P	$(M/2\log M)\frac{P}{3} = O(PM/\log M)$
Hypercube	P	$(M/2)\frac{P}{\log M} = O(PM/\log M)$
d-dim Toroid	P	$(2M^{(d-1)/d})\frac{P}{2d} = O(PM/\sqrt[d]{M})$
3-dim Toroid	P	$(2M^{2/3})\frac{P}{6} = O(PM^{2/3})$
2-dim Toroid	P	$(2\sqrt{M})\frac{P}{4} = O(P\sqrt{M})$
Ring	P	$P = O(P)$

Table 4.1: Characteristics of topologies used to connect M packaging modules

these messages is inversely proportional to the bisection width of the network[1]. Therefore, we can argue that the peak throughput of a network is proportional to its bisection width.

Table 4.1 shows a comparison of a few standard topologies for M modules, with P pins per module[2]. We can see that among the different topologies, the complete graph has the maximum bisection width for a constant pin-count. Therefore, we expect that the complete graph has the maximum peak throughput.

In fact, we can formally prove the following claim:

Claim 4.1 *When we have M modules with P pins each, and each module has an equal number of processors, the inter-module network that achieves the maximum peak throughput is a complete graph in which each edge of the complete graph corresponds to $\frac{P}{2(M-1)}$ wires. This network provides a peak throughput of $\frac{M^2P}{2(M-1)}$ bits per clock cycle.*

For the proof, we pose the problem of finding the network with highest peak throughput as a graph problem and show that the solution to the problem has the topology of a complete graph.

We model the inter-module network as a digraph $G = (V, E)$, with edge weights. We call G the *network graph*. Each node in the graph corresponds to a module, and each

[1] Note that our model of packaging technology assumes that the clock rate is independent of network topology(Section 2.3).

[2] All logarithms are to base 2.

edge corresponds to a connection between modules. The capacity of an edge e, denoted by $w(e)$ is the number of wires that correspond to that edge. Since we have M modules,

$$|V| = M$$

and since each module has a pin-count of P, for every module x

$$\sum_{y \in V} w(x,y) + w(y,x) = P$$

The communication traffic in the network is modeled as a complete digraph $G' = (V', E')$, with edge weights. We call G' the *communication graph*. Each node in G' corresponds to a processor, and each edge corresponds to the communication between a pair of processors. The weight of an edge $e' \in E'$, denoted by $p(e')$ is the probability that there is a message between the corresponding pair of processors. We use the communication graph to model one communication step, in which each processor sends exactly one message. In general, for each processor x

$$\sum_{y \in V'} p(x,y) = 1$$

Let the mappings $\phi : V' \rightarrow V$ and $\psi : E' \rightarrow E$ define an embedding of G' in G. The mapping ϕ is a many-to-one function, and specifies a partitioning of the processors into modules. A symmetric partitioning implies that exactly $\frac{N}{M}$ nodes of G' are mapped to each node of G. ψ is a one-to-many mapping since each edge in G' gets mapped onto some path in G, and ψ specifies the (multi)set of edges which form this path. Let ψ^{-1} be the inverse mapping of ψ; $\psi^{-1}(e)$ specifies the set of edges in G' that are mapped onto the edge e in G.

The total traffic through an edge e of G in one communication step, is denoted by $t(e)$ where

$$t(e) = m \sum_{e' \in \psi^{-1}(e)} p(e')$$

where m is the message length in bits. Since the edge e corresponds to $w(e)$ wires, the time taken to send the data across e is $\frac{t(e)}{w(e)}$ clock cycles. The time T taken to finish a single communication step is

$$\begin{aligned} T &\geq \max_{e \in E} \frac{t(e)}{w(e)} \geq \frac{\sum_{e \in E} t(e)}{\sum_{e \in E} w(e)} \\ &= \frac{\sum_{e \in E}(m \sum_{e' \in \psi^{-1}(e)} p(e'))}{(MP/2)} \end{aligned}$$

because $\sum_{e\in E} w(e)$ is the total number of wires in the inter-module network.

$$= \frac{\sum_{e\in E}(m/N)|\psi^{-1}(e)|}{(MP/2)}$$

because $p(e') = \frac{1}{N}$ for random traffic. Note that $\sum_{e\in E} |\psi^{-1}(e)| = \sum_{e'\in E'} |\psi(e')|$. Define the average path length to be $h = \sum_{e'\in E'} \frac{|\psi(e')|}{|E'|} = \frac{1}{N^2}\sum_{e'\in E'} |\psi(e')|$. Thus

$$T \geq \frac{(m/N)N^2 h}{MP/2} = \frac{2Nmh}{MP}$$

In each communication step, all the N processors send one message of m bits. Therefore for any graph the peak throughput is given by

$$\frac{Nm}{T} \leq \frac{MP}{2h}$$

The smallest value for h is $\frac{M-1}{M}$ because the maximum number of edges in $e' \in E'$ for which $|\psi(e')| = 0$ is $M(\frac{N}{M})^2$. This is because there are M modules with N/M processors each, and for any pair of processors that map onto the same module, $|\psi(e')| = 0$. This implies that $h \geq \frac{1}{N^2}(N^2 - \frac{N^2}{M}) = \frac{M-1}{M}$. Therefore, for any graph, the peak throughput cannot exceed

$$\frac{M^2 P}{2(M-1)}$$

In steady state the complete graph network with $w(e) = \frac{P}{2(M-1)}$ achieves this upper bound. In the complete graph all edges of E' that are not local to a module, are mapped onto only one edge in E, giving $h = \frac{M-1}{M}$. Further, $t(e)$ is the same for all edges, and making $w(e)$ the same for all edges ensures that $\max_{e\in E} \frac{t(e)}{w(e)} = \frac{\sum_{e\in E} t(e)}{\sum_{e\in E} w(e)}$. Therefore we have proved that the best inter-module network to connect M modules with P pins each, is a complete graph. □

Since the complete graph provides the best performance for a fixed cost, we choose the complete graph topology to interconnect the modules. We consider several intra-module network organizations for N=1024 processors. We also consider different module sizes, by choosing different values for s, the number of processors per module. We consider s = 16, 32 and 64, corresponding to M = 64, 32 and 16 respectively. For the sake of fairness, when we make comparisons between networks with different M, we hold the peak throughput constant. Note that holding the peak throughput constant, is equivalent to holding the bisection width constant, since the bisection width is equal to half of the peak throughput. For concreteness, we fix the bisection width at 8192, corresponding to a peak throughput of 16384 bits per cycle, or 16 bits per processor per cycle.

4.2.1 Scalability

It is somewhat surprising that we chose the complete graph to interconnect modules; complete graphs are considered not to be scalable as an interprocessor network. The rationale for this belief, is that they have a very large number $O(M^2)$ edges, with O(M) degree per node. We note however, that we wish the network design to be scalable for large values of N rather than M. The number of modules is not directly related to the number of processors: as we build larger machines, we will have to build them using a taller hierarchy, involving larger modules at the top level. If we are building a network for a 100,000 processors, it is unlikely that the top level of the packaging hierarchy will consist of modules that contain 10 processors each. The number of modules M at any level will depend upon the technology, but not substantially on the number N of processors.

We consider the issue of scalability at length in Chapter 7.

4.3 Intra-module networks

The only standard N processor network known to us that has a partitioning into M modules such that the interconnection between the modules is a complete graph is a Butterfly network. This requires M and N to be powers of 2 and $M = \sqrt{N}$. Figure 4.1(a) shows a butterfly network connecting 16 processors to 16 memories and figure 4.1(b) shows how this network can be partitioned into 4 modules, such that each module has 4 processors and 4 memories. Also note that the nodes in columns 2 and 3 have been reordered to make the connections between these columns local to the module. The numbers on the processors and memories are shown primarily to illustrate the reordering and do not carry any special significance.

Since we need to route access requests from the processors to the memories and the corresponding replies back from the memories to the processors, we need two separate butterfly networks. Therefore, each link in the figure should be interpreted as a pair of directed links. Note that the bundle of wires going between any two modules, A and B, actually consists of 4 different channels, two in each direction. One channel carries access requests from the processors in module A to memories in module B, another carries requests from B to A. Similarly two channels carry responses from A to B and B to A respectively.

Although we started with a *dance-hall* type network, it is useful to associate each memory with a processor from a practical point of view. This simplifies building the machine since each processor can use a a single memory that is partitioned into private memory and shared memory. While the private memory is accessed directly by the processor, the shared memory is accessible to all the processors via the network. We represent this in figure 4.1(c) by drawing the processors and their associated memories adjacent to each other.

Since the partitioning shown above is applicable when $M = \sqrt{N}$, we first consider the case $M = 32$. Later we will examine the cases $M = 64$, and $M = 16$.

4.3.1 M=32 Modules

Basic butterfly network

We consider a butterfly connecting 1024 processors to 1024 memories; partitioned into 32 modules in the manner of figure 4.1(b).

A bisection width of $B = 8192$ implies that pin-count per module is equal to $4B(M-1)/M^2 = 992$, for a total of $32 * 992 = 31744$ level-1 pins. Each pair of modules is connected by a bundle of 32 wires. Since each bundle has 4 channels, each channel is 8 bits wide. Since the inter-module channels are 8 bits wide, we get a uniform network if we make the intra-module channels also 8 bits wide. We call this network Butterfly.8.8. The first suffix stands for the intra-module channel width and the second suffix for the inter-module channel width.

The interconnections within a module are are shown in figure 4.2

Widening channels

The previous network used a uniform topology that did not take advantage of packaging locality. In general, at the lower levels of the packaging hierarchy we can have more wires and also transfer bits across these wires at greater speeds. Both these characteristics permit us to increase the capacity of the networks at the lower levels of the hierarchy. Instead of separately modeling increased wire density and clock rates, we consider networks that keep the clock rate constant and just use higher wire densities. Accordingly, we consider networks with wider intra-module channels: Butterfly.16.8 and Butterfly.32.8. Both networks have the same topology as Butterfly.8.8, but they have 16 and 32 bit wide intra-module channels respectively.

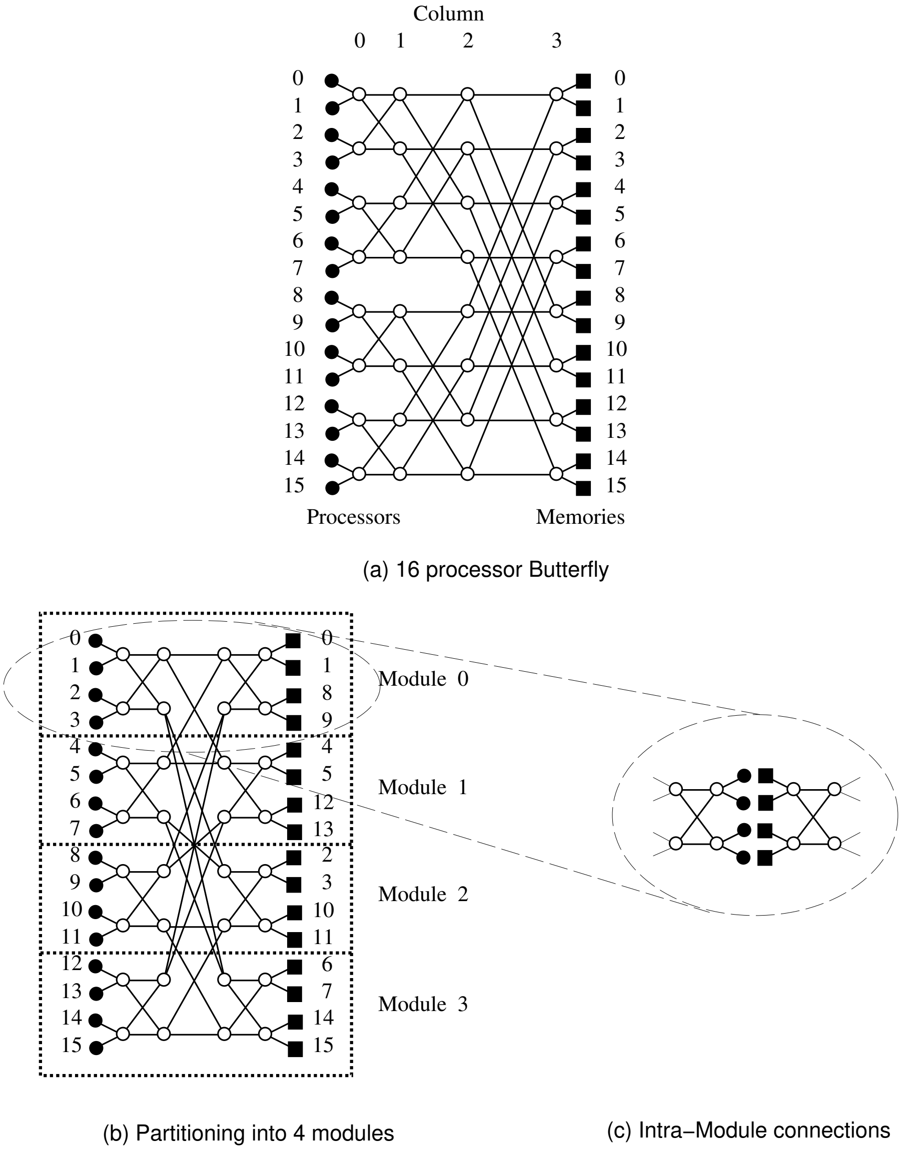

Figure 4.1: Butterfly Network and its partition

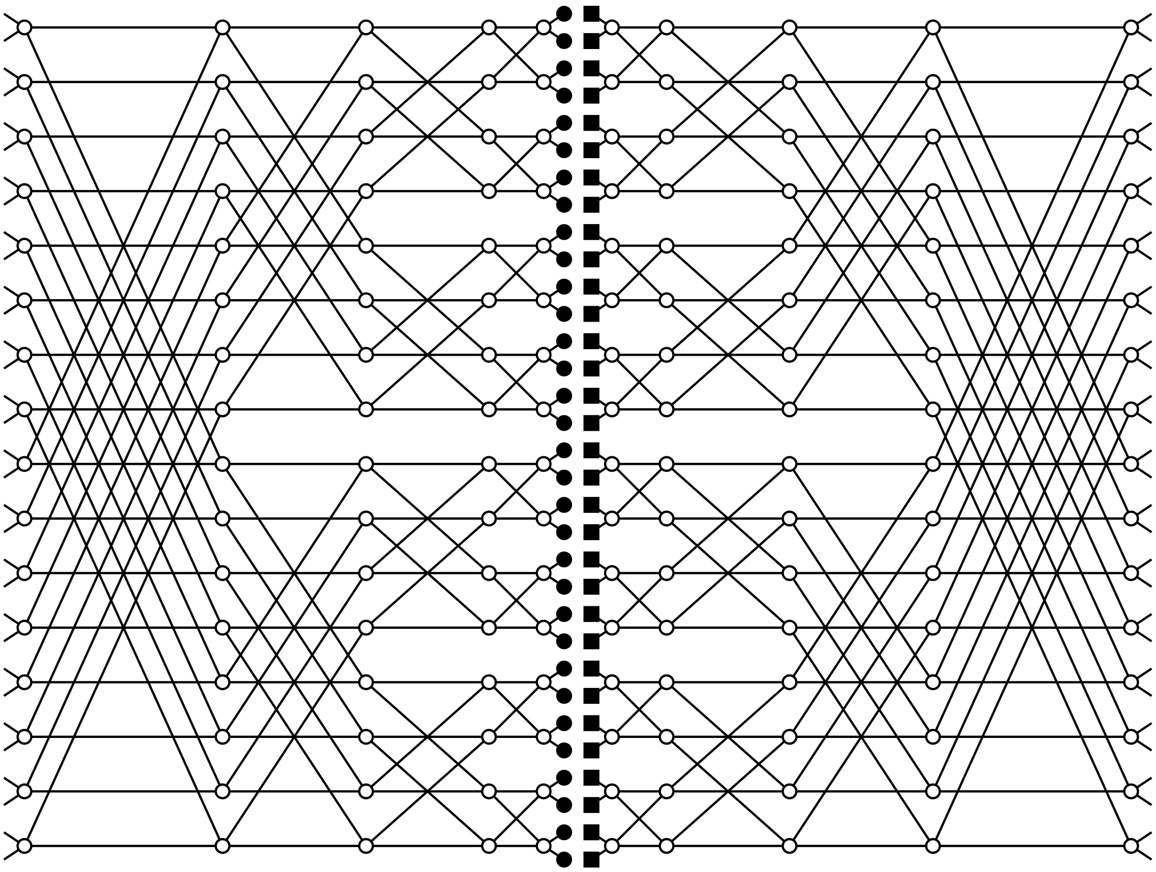

Figure 4.2: Intra-Module connections, Butterfly Network

It is important to point out that the costs of the these networks are different because they use different amounts of resources within a module. However, all of them have the same inter-module network, and if we assume that network cost is dominated by the cost of the top-level network, the difference in the costs of these networks will be small.

Dilation

Another method for utilizing more local wires is dilation. We consider networks Butterfly.8-2dil.8, Butterfly.8-4dil.8, and Butterfly.16-2dil.8. The first network has 2 8-bit wide channels between nodes, the second has 4 8-bit wide channels and the third has 2 16-bit wide channels. If there is more than one message that needs to go out in the same direction, they can be simultaneously sent along the dilated channels. The first network requires the same number of wires per routing node as Butterfly.16.8 while the second and the third use the same as Butterfly.32.8.

In the case of network dilation, we do not dilate the channels connecting into the memories or the processors. If we dilate the channels connecting to the memories, we may have to use multi-ported memories or add some complicated circuitry to prevent multiple accesses from being made simultaneously.

Multibutterfly

A 2-way dilated butterfly can be thought of as a pair of identical butterflies that are merged together. In the case of a multibutterfly [Upf89, LLM90] (or more precisely a *twin-butterfly*), the nodes within each column of one of the two butterflies are permuted in a special fashion before the two butterflies are merged. To generate a multibutterfly with I inputs, we take a collection of random permutations $\Pi = (\pi_0, \pi_1, \ldots, \pi_{\log I-1})$, where $\pi_x : [0, \frac{I}{2^{x+1}} - 1] \rightarrow [0, \frac{I}{2^{x+1}} - 1]$, and merge the node in row $\frac{jI}{2^{x+1}} + i$ and column x in the first butterfly with node in row $\frac{jI}{2^{x+1}} + \pi_x(i)$ and column x of the second butterfly, for all $0 \leq i \leq \frac{I}{2^{x+1}} - 1$, all $0 \leq j \leq 2^{x+1} - 1$, and all $0 \leq x \leq (\log I - 1)$.

A multibutterfly requires the same number of wires per routing node as a 2-dilated network, but the connections between the adjacent stages are more complex. Using random permutations in the wiring of these networks helps to avoid problems with non-uniformities in load distribution.

Note that the multibutterfly network described above is used only within the modules. Other network evaluation studies [LLM90, KU91] have considered connecting all the processors using a multibutterfly network. We do not consider such a network since we believe that the multibutterfly cannot be partitioned as well as a butterfly. Partitioning the multibutterfly into modules in a manner similar to a butterfly, necessitates a much larger pin-count per module. We suspect that any other type of partitioning of the multibutterfly also has the same problem.

Fewer memory units

When we increase the channel widths within the packaging modules, we increase the channel widths into the memory units, effectively increasing the service rate for memory access requests made over the network. In such networks, the effective memory bandwidth is greater than the network bandwidth. We observe that it might be possible to use fewer memory units when we have wider intra-module channels without appreciable degradation in performance. Reducing the number of memory units can potentially reduce the cost of the system, although it does not affect cost as per our coarse packaging model. A system with fewer memory units than processors corresponds to building a network using small shared memory multiprocessors as building blocks, instead of single processors.

We accordingly consider Butterfly.32.8.M-512 and Butterfly.32.8.M-256. Both networks have 32 bit wide intra-module channels and 8 bit wide inter-module channels, but the former has only 512 memory units and the latter has 256 memory units. Both networks have 1024 processors, implying that the former uses building blocks which are 2-processor shared-memory multiprocessors, whereas the latter uses 4-processor multiprocessors. Figure 4.3 shows the intra-module network for Butterfly.32.8.M-256. Since there are fewer than 32 memory units per module, there are fewer stages of switching connecting the memory units to the inter-module channels.

Dilating the inter-module network

We consider the network Butterfly.32.4-2dil which has 32 bit wide intra-module channels similar to Butterfly.32.8, but the channels of the inter-module network are 2-way dilated. When we dilate the inter-module network, we reduce the width of each channel by a factor of 2 to keep the pin-count per module the same. Therefore the inter-module channels in this network are 4 bits wide.

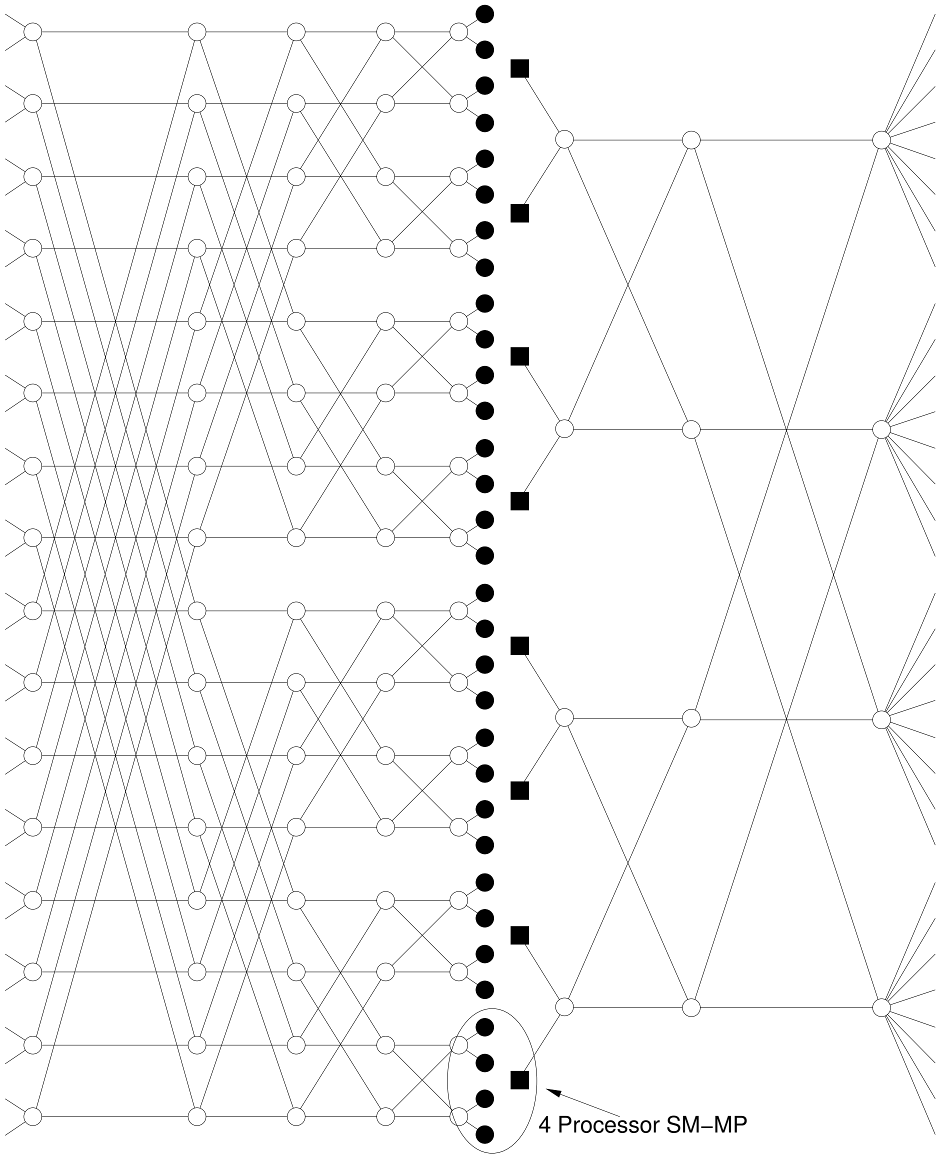

Figure 4.3: Intra-module network: Butterfly.32.8.M-512