CMU-CS-80-125

The Hashnet Interconnection Scheme

Scott E. Fahlman

June 2, 1980

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Abstract

This paper describes a type of switching network which can simultaneously connect many inputs to many outputs. Such networks are useful in communications, in dynamically reconfigurable computer systems, and in building semantic network memories for artificial intelligence and data base applications.

A *generalized connection network* (GCN) is a switching network in which each of N inputs can be connected to any of N outputs, many to one, with all of the N connections operating simultaneously. Such a network is said to be *non-blocking* if the connections can be established in any order, without rearranging existing connections and without any possibility that the establishment of an arbitrary new connection will be blocked by existing connections in the network. In *seldom-blocking networks*, there is some finite chance that a new connection will be blocked, but this chance can be made arbitrarily small through proper design of the network.

This paper will examine one class of seldom-blocking networks, which I call hashnets, with the emphasis on designing such networks for practical applications. These networks are built from layers of selector switches, with random hard-wired connections between the outputs of one layer and the inputs of the next. Such networks can give a close approximation to non-blocking behavior, but with much less hardware than the smallest known networks that are strictly non-blocking: the number of contact-pairs is $O(N \log N)$ rather than $O(N \log^2 N)$, and the constant factors are small. Hashnets are not an entirely new idea, but little has appeared in the literature about how to design and use networks of this kind.

This paper will explore the issues that arise in the design of practical hashnets, and will describe how to design a near-optimal hashnet for any desired combination of size and blocking probability. It will also present a scheme for time-sharing such networks, which greatly multiplies the number of (virtual) connections available with a given amount of hardware.

1. Introduction

A *permutation network* is a switching network with N inputs and N outputs that is able to establish one-to-one connections between inputs and outputs, with all N connections operating simultaneously. The input-to-output permutation thus established can be any member of a universe of N! possible permutations. A *generalized connection network* (GCN) is more general: it allows each of N inputs to be connected to any of N outputs, again with all connections operating simultaneously. In the GCN, a given input is connected to only one output at a time, but an output may be connected to many inputs at once. A GCN, therefore, selects one input-output mapping from a universe of N^N possible mappings. The terms "input" and "output" are used here merely to label the two sides of the network; depending on the application, the connections may be bidirectional.

Such switching networks are useful in many contexts. The most obvious use, and the one that has stimulated the most research in this area, is in meeting the switching needs of telephone systems [Benes 65]. In this application, the emphasis has naturally been on networks in which all connections are possible but only a small fraction of the connections are in use at once. In recent years, switching networks have been studied as a way of configuring parallel computer architectures to allow multiple processors to access multiple memories, terminals, and other resources in a flexible way with a minimum of contention over the communication channels [Masson 79, Goke 73]. For very small networks one can use a crosspoint switch to achieve this interconnection, but in such networks the number of contact pairs grows as N^2, too expensive for many large-network applications.

My own interest in generalized connection networks arises from my work in semantic network memories for artificial intelligence and knowledge base applications [Fahlman 79]. The precise requirements of such semantic network memories, along with a proposed design, are described in another paper [Fahlman 80]. For now it will be sufficient to note that a semantic network memory containing 10^6 elements (concepts and simple assertions), enough for moderate expertise in many real-world domains, will require a GCN with 4×10^6 inputs and 10^6 outputs. Such a network would be quite costly to build by conventional techniques.

Switching networks can be divided into three classes according to how the connections are made. *Blocking networks* guarantee that any single connection can be made, but do not guarantee that all legal combinations of connections ("legal" according to the definition of a GCN or permutation net) can be made at once. In other words, some legal connections may block others from being present at the same time. *Rearrangeable networks* guarantee that any legal combination of connections can be set up in the network, but only if existing connections can be rearranged to accommodate any new request for a connection. *Non-blocking networks* are the most powerful, guaranteeing that any new request can be satisfied without the need for any rearrangement of existing connections.

It is possible to construct non-blocking GCNs, but they are expensive: the best known solutions for large N require $O(N \log^2 N)$ switch contact pairs [Pippenger 78a]. Furthermore, in such networks the delay through the network (that is, the number of switches through which a signal must pass) is $O(\log^2 N)$. It is known [Pippenger 78a] that non-blocking networks of $O(N \log N)$ contacts can be built, but the proof is non-constructive and no practical implementation strategy is known.

Rearrangeable networks with $O(N \log N)$ connections and $O(\log N)$ delay are known. Ofman [Ofman 67] has demonstrated that a GCN can be built with $8N \log_2 N - 6N$ contact pairs with a delay of $4 \log_2 N - 3$. Thompson [Thompson 78] has improved slightly on these figures to produce a GCN with less than $7.6N \log_2 N$ contact pairs. Such networks can be configured for a given set of connections by a procedure due to Waksman [Waksman 68] which takes $O(N \log N)$ time to configure the network. Unfortunately, this set-up procedure must be repeated whenever a new connection is added to the network, and the set-up does not appear to be amenable to parallel solution. This costly set-up procedure is the principal reason why the rearrangeable networks have not been used in many applications for which they would otherwise be appropriate. In multi-processor architectures, for example, there has been more interest in the simpler Banyan networks [Goke 73], which can be configured in $O(\log N)$ time, but which block on many of the potential mappings.

The above analysis assumes networks that are *strictly* non-blocking or rearrangeable. In many applications, it is possible to get the best of both worlds through the use of seldom-blocking networks [Pippenger 78b, Pippenger 75]. These networks behave as non-blocking networks in almost all cases, but there is some small but finite chance that a desired connection cannot be made. By choosing an appropriate network configuration, it is possible to make the probability of failure arbitrarily small. One kind of seldom-blocking network, first described by Marcus [Marcus 72], is built from selector switches arranged in layers, with a random pattern of interconnections between successive layers. Because this scheme is analogous in many ways to a hash-coded data structure, I propose the name "hashnets" for networks of this sort.

In some applications any chance of blocking, however small, may be unacceptable. Where seldom-blocking networks can be used, however, the savings can be substantial: Marcus [Marcus 72] has shown that a hashnet can be constructed with $O(N \log N + \log 1/P_B)$ contact pairs, where P_B is the allowable probability of blocking in the attempt to establish any single connection. Using a hashnet we can obtain nearly the effect of a non-blocking network for roughly the cost and delay of a rearrangeable network, a cost which is close to the information-theoretic minimum. As we will see, the decision to accept some very small risk of not finding a desired connection also makes it possible to effectively time-share these networks, dramatically increasing the number of virtual connections available with a given amount of hardware, at the cost of reduced bandwidth in the individual connections.

Such networks are not as well known in computer science circles as they might be, perhaps because their treatment in the literature has been primarily theoretical and not oriented toward practical applications. This paper describes the hashnet scheme in detail, and explores how certain design parameters can be varied to obtain a network of any desired size and probability of blocking while minimizing cost.

Section 2 of this paper will develop the basic hashnet scheme in detail, and will examine the design of a 1000 x 1000 GCN. Section 3 will describe how new connections are found within such a network. Section 4 will describe how such networks can be time-shared to greatly increase the number of connections for a given amount of logic, at the expense of bandwidth in the individual connections. Section 5 will contain some miscellaneous points and observations that may be of use to potential users of the hashnet scheme.

2. The Basic Hashnet Design

The basic element in a hashnet is the *selector cell* shown in figure 1.This is simply a selector switch which can connect its single input to any one of F outputs. The parameter F is constant for all cells in the network, and is referred to as the *fanout* of the cell. F is usually chosen to be one less than some power of 2, so that the state of the cell can be compactly represented in a few bits of memory, with one value reserved to indicate that the cell is currently unused. In most of the examples we will see, F will be 7 or 15.

We will now consider how to build a hashnet out of these cells. We will begin with the design of an NxN permutation network, and will see later how the same network can be set up as a GCN. Figure 2 shows the cells arranged into L layers with 2N cells in each. Each input terminal is wired to the inputs of two cells in the first layer. Each of the interior cells in the network has its F outputs wired to F *randomly-chosen* cells in the next layer. (Note that, on the average, F wires will be tied to a cell's single input terminal.) These random connections are hard-wired when the network is built; all of the dynamic configuration of the network is done by setting the state of individual cells. Each output terminal receives, on the average, 2F wires from the final layer of cells.

Each cell participates in only one connection at a time. Thus, in a permutation network with N connections running through it, half of the cells in each layer will still be unused. This excess capacity is of critical importance in a non-blocking network; there must be some slack if the last connection is to be made without disturbing the N-1 connections already present in the network. The factor of 2, while convenient in many cases, is just another parameter of the network. In general, we will use use layers of KN cells each, where K is greater than 1 and seldom more than 2. We will also use the symbol U to represent the fraction of cells in a layer that are *unused* after the last connection

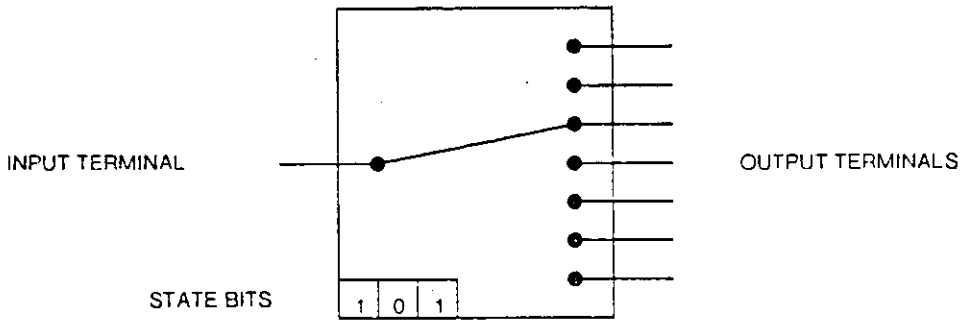

Figure 1: Selector Cell

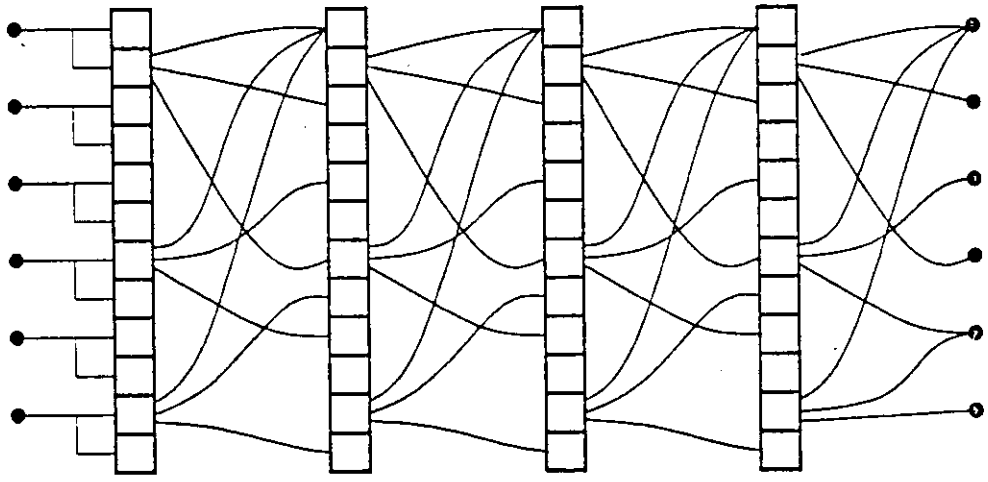

Figure 2: The Basic Hashnet Arrangement
(Some wires omitted for visibility.)

is made. Note that U = 1 - 1/K, so if K is 2, U is 0.5; if K is 1.5, U is 1/3.

We can now begin to look at the statistical behavior of these networks. Let us assume a network with 1000 input terminals, 1000 output terminals, 2000 cells in each internal layer, and a fanout of 7. Suppose we begin with an empty network and pick an input terminal and an output terminal at random. How does the probability of finding a connection between these terminals depend on the number of layers and the other parameters of the network? We begin with connections to two cells in the first layer. With a fanout of 7, each of these cells has a potential connection to 7 randomly-chosen cells in the next layer. Let us assume, for now, that the cells to which these wires connect are all distinct. In this over-simplified model, we have a choice of 14 destination-cells in the second layer, 98 in the third, and so on. After $\log_7 N$ layers, we reach a layer in which every cell has a potential connection to the chosen input. We replace these cells with output terminals and, for whichever output terminal we specify, the chance of finding a path is 100%.

Of course, the assumption that we made above, that the branches in the tree of reachable cells do not intersect one another, is not true in the actual network. There will be few collisions in the early layers of the network, when the set of reachable cells is a small fraction of the KN cells in the layer, but as we add more layers the number of reachable cells approaches saturation and collisions become significant. A more accurate model is obtained by noting that if M cells are reachable in one layer of the network, the number of wires going from them to the next layer is MF. These wires are distributed randomly over the KN possible destination cells. If we represent the probability of a destination cell *not* being reachable as P_B, we get for large N:

$$P_B = e^{-MF/KN}$$

If M' is the expected number of reachable cells in the next layer, we get:

$$M' = KN(1 - P_B)$$

When M is small compared with KN, M' is close to MF. As M approaches KN, the probability of not being able to reach a given destination cell approaches the asymptote of e^{-F}, and adding additional layers does no good. Thus, if there are "enough" layers in the network, the chance of being unable to make a desired connection is governed only by the fanout of the selector cells and the number of cells to which the input and output terminals are connected. By working out a few examples, we can see that for fanouts of 7 or more, "enough" layers means one or two more than the $\log_F N$ layers that we need in the ideal, non-colliding model. The precise number is hard to characterize analytically; it is a function of the fanout, of how close to the asymptote we want to get, and of exactly where $\log_F N$ falls in the interval between integers corresponding to the layers of the network.

Table 1 gives some values for a network with an N of 1000, a K of 2, and a fanout of 7. The number of cells indicated is the expected number in each layer that can be connected to any particular input.

The "probability of losing" figure gives the chance of not being able to make a desired connection if the network is terminated after the specified number of layers. This number takes into account the fact that there are twice as many cells in each interior layer as there are output terminals -- that is, each output terminal has 14 wires coming into it rather than the usual 7. This improves the odds of finding a path substantially. For now, ignore the rightmost two columns of the table. Table 2 gives the same figures for a fanout of 15 in each cell.

Under the conditions described above, we note that with a fanout of 7 we need 5 layers to get a low (10^{-6}) probability of blocking, and that adding more layers does not improve things substantially over that figure. With a fanout of 15, we need 3 layers to reduce the chance of blocking below 1%, and adding a fourth layer causes the chance of blocking to drop below 10^{-13}.

All of the above figures have assumed that the network is empty. If the network is indeed non-blocking, we must have a high probability of finding the Nth (and final) path through the network even when N - 1 paths are already in use. In this case, the calculations are identical to those above, but at each layer we must multiply the number of reachable cells by U, the fraction of cells that are not busy when the network is already holding N - 1 connections. This is roughly equivalent to reducing the effective fanout of the cells from F to UF.

The rightmost two columns of tables 1 and 2 give the number of reachable cells and the chance of blocking at each stage when we are adding the *last* of N connections to the networks studied above. This assumes that the connections already in place are not to be moved. Note that with a fanout of 7 we now must go to 6 layers to get the chance of failure under 2%, and that with a fanout of 15 we must go to 4 layers to get a .03% failure rate. Depending on the size of the failure rate that we can tolerate, we may have to increase the fanout or the K factor, but the network still grows only as O(N log N) and the delay as O(log N).

There is an additional source of possible failure that is not reflected in the model above. In the first and last layer of cells, where the tree of reachable connections is still rather narrow, there is some chance that *all* of the cells reachable from a given input or output will be busy carrying other connections. With a fanout of 7 and a K of 2, the chances of such an event are about 2^{-14} or 10^{-4} at each end, a small number compared to the asymptotic chance of blocking that we computed above for the Nth connection. With a fanout of 15, we get 2^{-30} or 10^{-9}. Once the tree has spread to interior layers, the number of reachable cells becomes so large that the chance they will all be blocked becomes negligible. Note, however, that a deviation from true randomness in the network's inter-layer connections could cause congestion in certain regions and increase this figure substantially.

To summarize, in designing a hashnet to serve as an NxN permutation network we first choose

Table 1

Fanout	7
Terminals	1000
Output Terminals	1000
Cells / Layer	2000
Fraction of Cells Used	.5

	In Empty Network:		In Full Network:	
Layer	Cells Reached	Prob. of Blocking	Cells Reached	Prob. of Blocking
0	2	0.998000994	2	0.998000994
1	14	0.98609752	14	0.98609752
2	95	0.90695908	48	0.95234399
3	565	0.51315154	161	0.71634594
4	1723	0.0188516192	491	0.137301195
5	1995	5.68441296E-6	1153	2.38419962E-3
6	1998	8.5974574E-7	1734	9.2722844E-4
7	1998	8.4239745E-7	1903	9.1781995E-4
8	1998	8.4228804E-7	1928	9.1775945E-4

Table 2

Fanout	15
Input Terminals	1000
Output Terminals	1000
Cells / Layer	2000
Fraction of Cells Used	.5

	In Empty Network:		In Full Network:	
Layer	Cells Reached	Prob. of Blocking	Cells Reached	Prob. of Blocking
0	2	0.998000994	2	0.998000994
1	29	0.97044553	29	0.97044553
2	391	0.63977298	206	0.79985812
3	1893	2.46822253E-3	1076	0.0496812095
4	2000	4.20163906E-13	1964	6.44511867E-7
5	2000	9.3314245E-14	1998	3.05904567E-7
6	2000	9.3314245E-14	1998	3.05904567E-7
7	2000	9.3314245E-14	1998	3.05904567E-7
8	2000	9.3314245E-14	1998	3.05904567E-7

values of F and K that will give an asymptotic blocking probability in the desired range. Then we choose the number of layers to approach this probability as closely as is desired. Useful values are K = 2, F = 7 or 15, and L = $\log_{UF}N$ + c, where c is equal to 1 or 2.

3. Finding New Connection Paths

So far, we have only considered whether a free path exists for a desired connection and not whether such a path can easily be found. In fact, a rather simple parallel procedure can be used to find an available path if there is one. We have seen that each cell contains a few bits of memory in which it records whether it is in use and, if so, which output wire is connected to the input. To this we will add another bit that is used for temporarily marking the cell in question.

Now, suppose we have been given an input and an output that are to be connected. We begin by putting some sort of signal on the specified output terminal. All of the non-busy cells in the last layer that can see this signal on any of their output wires are then marked. These cells then signal to the cells in the previous layer that can see them, and so on. (See figure 3.) Carried to completion, this process marks all the non-busy cells in the network that can make connections to the desired output. As we have seen, it is very probable that these markers will reach at least one of the first-layer cells connected to the specified input. (If this is not the case, the desired connection cannot be made without rearranging the network.) Now we simply backtrack, moving from the input to the output, at each stage selecting one of the marked cells in the next layer and setting up that connection. By following a path of marked cells back to the output side, we are sure of reaching the desired output terminal. If the wires are truly scrambled after leaving each cell, the cell's internal logic can simply select the (internally) lowest-numbered output wire whose destination cell is marked -- this will result in a random distribution of connections going to the next layer.

Surprisingly, the same network that implements an NxN permutation net can, with minor modifications to the connection-finding procedure, be used as an NxN generalized connection network as well. Suppose that the network already contains N input-to-output connections. It does not matter whether these go to distinct outputs or are many-to-one; the essential point is that at most N of the KN cells in any intermediate layer are busy, just as before. If we select an output terminal with no pre-existing connections, we can mark a tree of idle and reachable cells back to essentially all of the input terminals, just as before. Now, instead of selecting just one input terminal and one path back to the output, we can select and connect as many inputs as we like. The connections may merge at the input of some intermediate cell in the tree, or they may not merge until the output terminal is reached, but the effect of wiring together all of the specified inputs and the selected output is achieved. This is a party-line connection; it is up to the user of the network to establish protocols to prevent confusion of signals in this connected subsystem.

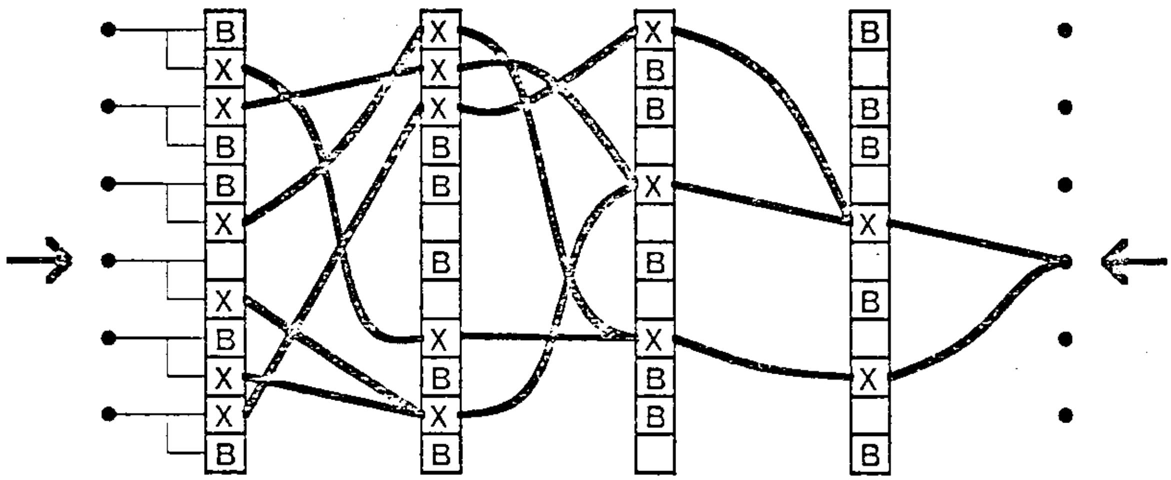

Figure 3A: Mark Back From Output
B = Busy, X = Marked

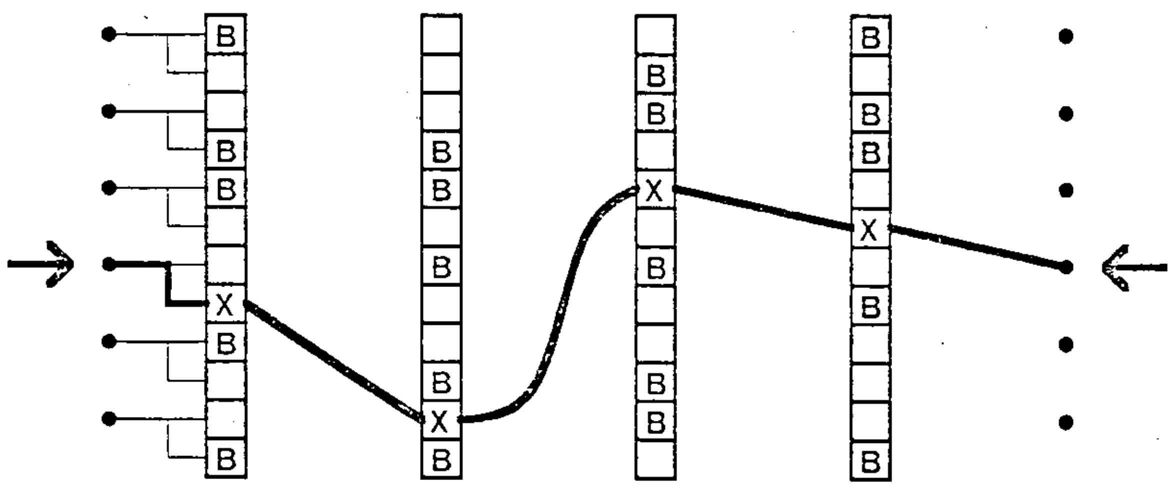

Figure 3B: Select One Marked Path

It is not necessary to make all of the connections to a given output terminal at once. We want to be able to connect a new input to an output that already has some input connections. The problem is that the pre-existing connections to this output might tie up most or all of the incoming wires and immediately adjacent cells. The fix is simple, however: instead of insisting that the new connection use only free cells, we allow it to join the tree of existing connections to the specified output at any point. To do this, we mark back from the output as before, but this time we mark both idle cells that can form connections to the specified output and busy cells that are already tied to this output. Then we trace a path back from the desired input along marked paths, just as before. This will not bother neighboring paths at all; in fact, this procedure will result in the use of fewer intermediate cells than are used in the permutation network, since some branches of the tree will carry the connections from several inputs.

4. Time-Shared Hashnets

Using the internal-merge technique that will be described in the next section, it is possible to pack 15 of the 15-way selector cells onto a single IC chip. (The limiting factor is not the capacity of the silicon chip itself, but rather the limited number of wires that can be attached to a chip.) This means that a 1000 x 1000 GCN can be built with about 550 chips -- a rather small system by current standards. But the million-connection networks that we need for semantic network memories (and many other uses) would require about a million IC's. A system of this size is out of the question for most applications; in fact, it would probably be impossible to maintain even if cost were no object.

A million-connection network can be built, however, by using a 1000 x 1000 network and time-sharing it 1000 ways. The price paid for this, of course, is that each virtual connection has only 1/1000 the bandwidth of the network's real connections. If 1000 one-bit signals can be moved through the thousand-connection network in a microsecond, it will take roughly a millisecond to move one million one-bit signals through the million-connection time-shared network. It happens that this reduced bandwidth is acceptable for semantic network memories and probably also in many applications involving audio-frequency transmission.

To see how the time-sharing works, we must first note that each of the 1000 inputs and outputs of the network is now time-multiplexed 1000 ways. Instead of having a terminal all to itself, a user of the network is assigned to a particular terminal and to a particular time-slice from the set of 1000. We assume that all network users (senders and receivers) have access to the same clock signal, so they always know when it is their turn to talk or listen. The assignment of time-slices is permanent, just like the connection of users to physical terminals. The time slices roll by in a circular fashion, 0 to 999, then back to 0 and around again.

During each of the 1000 time-slices, the physical hashnet is set up differently. The random wiring is, of course, constant over all time slices, but the selection made within each cell is different for each time-slice. What this means is that instead of having just four bits of internal state in each cell, we have a 4 x 1000 circular shift register. During each time-slice, a new 4-bit value is shifted into the cell, and is used during that time to gate signals through the cell to one of the 15 possible outputs. A value of 0 marks the cell as unused during that time-slice.

With these changes, we have made it possible for arbitrary inputs and outputs to communicate with one another, but only if they are assigned to the same time slice. To get to a million-connection GCN, we also need some mechanism for moving signals from one time-slice to another. For single-bit signals, such a time-shifter is easy to build. (See figure 4.) During each time slice, we accept an input bit and a ten-bit address giving the time-slice that this bit is supposed to be moved to. This address is used to store the bit into the appropriate location of a 1000 bit random-access memory. After all 1000 slices have rolled by, this memory is fully loaded. During the next cycle of 1000 time-slices, we step through sequential locations in this memory and send out the stored bits in their new order. By using two 1000-bit memories, we can pipeline this system, loading one memory while unloading the other. Each of these time-shifter elements, then, contains a 10 x 1000 bit shift register and 1000 or 2000 bits of random-access memory.

Since each time-shifter that a signal goes through introduces a delay, it is desirable to use only a single layer of shifters. The probability of blocking is minimized if we put this layer of shifters in the middle of the network. We will need 2000 shifters, one at the input of each of the 2000 cells in the third layer. (See figure 5.) Note that with 2000 cells and 1000-way time slicing, we have two million virtual shifter inputs and an equal number of outputs. Even with one million connections already in place, only half of these inputs and half of the outputs (randomly distributed) will be busy. A busy input, for a given time slice, is indicated by a value of 0 in the corresponding slot of the 10-bit shift register; an output is busy if the third-layer selector cell to which it is attached is busy during that time-slice.

Now, what are the odds of being able to find a connection from one input on one time slice to one output on a different time-slice? If we consult table 2 for the network with fanout of 15 and 4 layers, we see that after two layers of cells with the network full, the typical input terminal can see 206 of the 2000 shifters. On the average, half of these will be busy during the time slice of the input in question. Therefore, about 100 non-busy shifters are available for forming the connection. By the same reasoning, each output terminal can see about 100 shifters on its time-slice. If these two sets of 100 shifters (chosen from a universe of 2000) have a non-null intersection, then a path can be found. For any single shifter in the input set, the chance of being in the output set is 100/2000 = .05 and the chance of losing is .95. But with 100 chances to win, the probability that they will all lose is $.95^{100}$ =

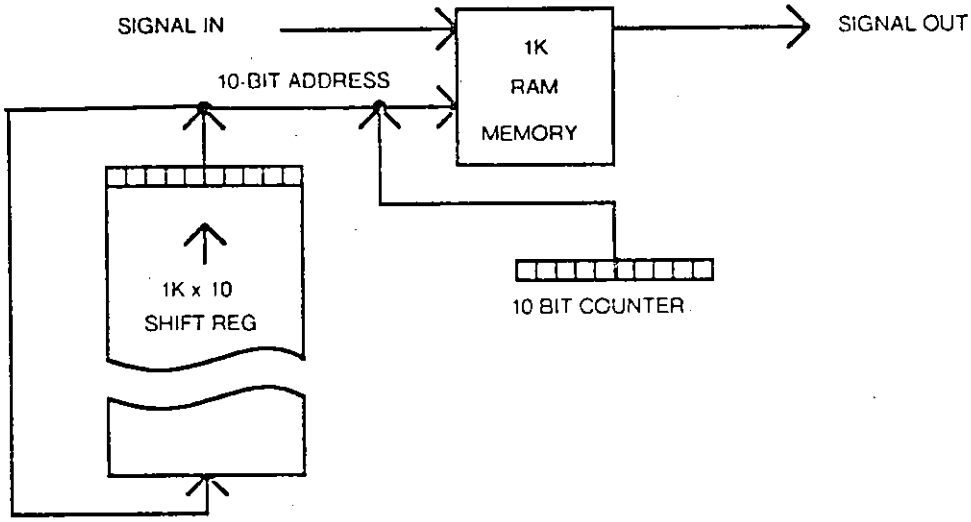

Figure 4: The 1K Time Slice Shifter

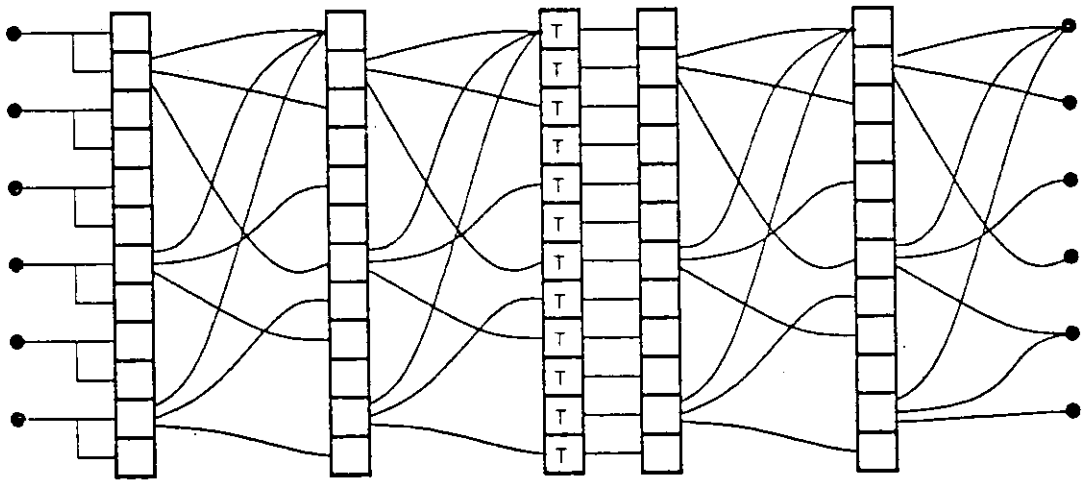

Figure 5: Time-Shared Hashnet
(T = 1K Time Slice Shifter)

.006 -- less than 1%. If this figure is too high, it is possible to add one extra layer of cells to the network reducing the probability of blocking to about 10^{-12}. We can place this new layer on either side of the shifter layer -- the effect is the same.

The path-finding algorithm is essentially the same as in the non-timeshared networks. We begin with the desired output terminal and time-slice, and we mark backwards through the network on that time-slice until we reach the layer of shifters. We mark all those shifters whose ouput is not busy on this time-slice and who can establish a connection to the desired output terminal. We then consider the time-slice of the desired input. If the shifter is busy on that slice, the mark is removed; otherwise, it is propagated back toward the inputs on the input time-slice. When the proper input is reached (on the proper time-slice), a single trail of marked cells is followed back to the output and the proper settings are written into the shift register memories of these cells. In effect, the time-shifter can be treated as a selector cell whose fanout is 1000 rather than 15.

While I have been using a figure of 1000 time-slices for the purposes of illustration, there is certainly nothing magic about this number. In general, any number of time-slices can be used. The bandwidth of the resulting connections is reduced proportionately, and the length of all the shift registers grows linearly with the number of slices. This kind of growth can be traded off against the N log N growth in cells and wires that occurs when the size of the physical hashnet is increased. This flexibility is useful in designing a system to fit existing memory parts and other engineering restrictions.

5. Concluding Remarks

Note that in the heavily time-shared networks, most of the cost is in the various shift register memories. In the four-layer million-connection design, the network requires 54 million bits of storage: 8000 selector elements, each with 4000 bits of shift register, plus 2000 time shifters, each with 10000 bits of shift register and 1000 bits of RAM. (Several selectors or shifters can be packed onto a single IC chip.) Note that the information-theoretic lower bound on the amount of state in such a network is 20 million bits: 10^6 connections, each of which requires 20 bits to specify which of the 10^6 possible destinations it goes to. Therefore, we are within a factor of three of the simplest possible design, and we get 1000-way parallelism from the hashnet.

Note too that in many technologies, shift register memories are the cheapest kind to build. Since the time-slices roll by steadily, it is possible to use dynamic memory chips rather than the more expensive static memories. Charge coupled devices or magnetic bubble memories may be attractive for very large scale (but rather slow) hashnets.

My most recent design for the million-element semantic network memory, which contains a GCN of

4 million connections, uses a 1000 x 1000 hashnet time-shared 4000 ways. The network portion of this machine contains about 4800 chips. 4000 of these are commercial 64K RAM chips, and the remainder are custom-designed selector and time-shifter chips. Again we are about a factor of 3 from the information-theoretic minimum of 80 million bits. Details of this design can be found in [Fahlman 80].

Two design tricks are used in the above design to reduce the number of IC packages. Since these appear to be generally useful in hashnet designs, I will describe them here. The first of these tricks depends upon the observation that the random pattern of interconnection between layers of the hashnet can be the same for each pair of layers. I have not been able to prove that this is true, but in a number of network simulations this change has made no measurable difference in the probability of blocking. This property makes it possible to use only one layer of selector cells, with the outputs tied through a random pattern of wires back to the inputs. First the signal bits are shifted into this layer of cells. Then the selection bits for the first layer are shifted in, and the signal bits are sent through the selected output wires and back to the inputs where they are latched. Next, the selection bits for the second layer are shifted in, and the process is repeated; this continues for the desired number of "layers". This technique does not reduce the number of bits of memory in the machine, but it does reduce the number of selector cells and the amount of random wiring by a factor of L, the number of layers of selector cells. Obviously, this is somewhat slower than the normal straight-through scheme, and it prevents any pipelining of signals.

The second trick involves the packing of 15-way selector cells onto IC chips. The limiting factor is not the silicon chip itself, but the number of external connections that can be made to a chip with current packaging technology. At present, 64-pin packages are the largest that are readily available, and smaller packages are considerably less expensive. Since each selector cell has one input and 15 outputs, and since 6 or so wires are needed for power, clock, and control signals, it would appear that only two or three selector cells can be placed in a single package.

The trick is to place 15 cells in a single package and to tie together their outputs within the chip, as shown in figure 6. Now each input pin is wired to exactly one output pin from the previous layer; in effect, we are moving the merger that used to occur at the selector cell input pins back into the cell packages in the previous layer. This alters our original assumption that the interconnections are totally independent of one another, since now if none of the outputs of a given cell can see the desired output terminal, there are 14 other cells in the same layer with the same problem. On the other hand, this scheme eliminates the problem of some cells getting more than 15 incoming wires while other cells get less than their share. I have not been able to determine analytically what difference this change makes in the probability of blocking, but we have simulated the network both ways and the 15 x 15 packing scheme makes no measurable difference. With this scheme, a 15 x 15

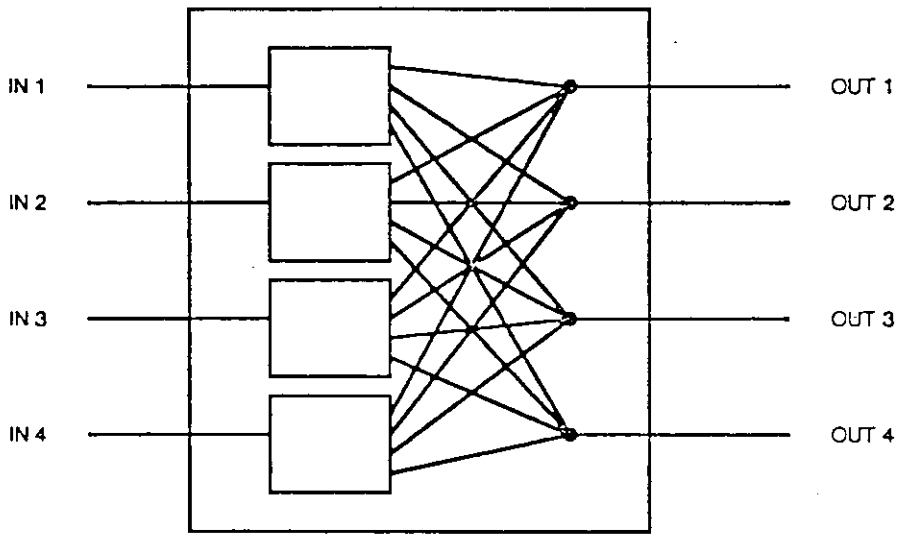

Figure 6: Selector Cells with Internal Merge
(Drawn as 4x4 instead of 15x15)

selector fits easily into a 40 pin package. There seems to be no problem with using both of the above tricks at the same time.

Hashnets have some curious and possibly useful reliability properties. If a few dozen selector cells in interior layers of the network are destroyed, this will alter the probability of blocking very little. Of course, if these cells are already carrying network connections, these connections will be broken (forgotten) and will have to be re-built. It is important that the damaged cells fail in the proper way, refusing to make connections rather than making spurious ones. If this failure mode can be enforced, it may point the way toward full-wafer integration, in which an entire hashnet is placed on a large silicon wafer. Such wafers always have a number of localized faults, but in a hashnet these might not matter.

The resemblance of non-timeshared hashnets to the layered structure of the neocortex of the brain is not altogether coincidental. The hashnet idea came to me while I was thinking about some diagrams of neural tissue. Here were regular layers of switching devices connected by seemingly random wiring between the layers, with many millions of cells in each layer and a fanout of perhaps 1000 or more. What were the odds of finding a specific connection through such a mess? Or a few million connections simultaneously? The time-sharing idea occurred later, and I found the earlier work of Marcus and Pippenger later still. I am not prepared to claim that any part of the brain is, in fact, a hashnet -- I have neither the training nor the facilities to investigate such a hypothesis -- but

hashnets may serve as an interesting if over-simplified model. The reliability properties noted above would certainly be useful in a machine made of neurons.

One interesting question is whether the randomness of the wiring in a hashnet is really essential or even beneficial. Could not some more orderly scheme do just as well and perhaps be less expensive to build? I do not know the answer to this. The randomness in a hashnet plays roughly the same role as the randomness in a hash table: it scatters the pattern of usage more or less evenly through the network, even if the input has some very regular pattern. In this way, it prevents unforseen congestion in parts of the net. For any given pattern of inputs, a structured scheme could do better, but this scheme might do much worse for some other set of inputs. In any event, if we abandon the assumption of randomness and independence of the connections, the prediction of network performance becomes very much more difficult. This question deserves additional study.

Acknowledgements

Gerald Sussman, Jerry Feldman, H. T. Kung, Clark Thompson, Nicholas Pippenger, and MACSYMA contributed time, ideas, and reference materials to this work. David Touretzky and John McDermott made many valuable suggestions to improve the clarity of this paper. The network simulations were programmed and run by Leonard Zubkoff.

Bibliography

[Benes 65] Benes, V.
Mathematical Theory of Connecting Networks and Telephone Traffic.
Academic Press, New York, 1965.

[Fahlman 79] Fahlman, S. E.
NETL: A System for Representing and Using Real-World Knowledge.
MIT Press, Cambridge, Mass., 1979.

[Fahlman 80] Fahlman, S. E.
Preliminary Design for a Million-Element NETL Machine.
Technical Report, Carnegie-Mellon University, Department of Computer Science, 1980.

[Goke 73] Goke, L. R. & Lipovski, G. J.
Banyan Networks for Partitioning Multiprocessor Systems.
In *Proceedings, First Annual Computer Architecture Conference.* 1973.

[Marcus 72] Marcus, M. J.
New Approaches to the Analysis of Connecting and Sorting Networks.
Technical Report 486, MIT Research Laboratory of Electronics, 1972.

[Masson 79] Masson, G. M., Gingher, G. C. & Nakamura, S.
A Sampler of Circuit Switching Networks.
IEEE Computer , June, 1979.

[Ofman 67] Ofman, J. P.
A Universal Automaton.
Transactions Moscow Mathematical Society 14, 1967.
Translation published by Amer. Math. Soc., Providence, RI, 1967, pp 200-215.

[Pippenger 75] Pippenger, N.
On Crossbar Switching Networks.
IEEE Transactions on Communications Com-23(6), June, 1975.

[Pippenger 78a] Pippenger, N.
On Rearrangeable and Non-Blocking Switching Networks.
Journal of Computer and System Sciences 17(2), October, 1978.

[Pippenger 78b] Pippenger, N.
Complexity Theory.
Scientific American 238(6), June, 1978.

[Thompson 78] Thompson, C. D.
Generalized Connection Networks for Parallel Processor Intercommunication.
IEEE Transactions on Computers C-27(12), December, 1978.

[Waksman 68] Waksman, A. A.
A Permutation Network.
Journal of the ACM 15(1), January, 1968.