1. Adding an FPX to each port of the WUGS switch allows it to function as an IP router with 1 Gb/s links. How many packets per second must be processed by the IP route lookup mechanism to handle minimum size IP packets? The route lookup mechanism shares one of the two SRAMs with the exact match classifier. If one third of the SRAM bandwidth is allocated to the route lookup mechanism and two thirds to the exact match classifier, what is the maximum number of memory accesses that can be performed for each IP address lookup. Is this number sufficient, do you think (the address lookup mechanism uses a 4 bit stride).

A minimum TCP packet is 40 bytes. This is commonly used as a benchmark packet size for IP routers. With the ATM interfaces used in the WUGS, the true minimum is once cell or 53 bytes. At 1 Gb/s, we can receive 2.4 million packets per second.

Since the SRAM clock rate is limited to 100 M Hz, one third of the memory bandwidth is 33 M Hz, so we can do 33 million memory accesses per second, each of which is 36 bits wide. This means that we can do 13 memory accesses for each lookup. With a 4 bit stride, the IP lookup data structure needs at least 8 memory lookups in the worst-case. The true number is a little higher (11), so the 13 available accesses is sufficient.

2. Consider the SPC performance numbers discussed in the notes (for IP packet forwarding). How much better performance do you think can be expected from the SPC-2. Keep in mind that the SPC-2 has a faster main memory interface and a faster CPU clock rate. However, it has the same PCI bus as the SPC1 (32 bits wide, 33 MHz).

The instruction processing rate is about 3 times faster, and the memory bandwidth is about 50% faster larger. With minimum size packets, the SPC-1 can process a little more than 100 K packets per second, meaning it is using about 85 M b/s of the PCI bus bandwidth of approximately 1 Gb/s. So the PCI bus bandwidth should not be the limiting factor. The memory bandwidth usage is at least twice the PCI bandwidth usage (because of the word-swap bug) and for minimum size packets, it’s more like three times, since the processor must access memory to read the header and do the address lookup. Even so, the memory bandwidth usage is only about 250 M b/s, while the total memory bandwidth for SPC-1 is 64*67 M Hz or roughly 4 Gb/s. So, it would seem that the memory bandwidth should not be a limiting factor either. This suggests that for minimum size packets, the packet processing rate could increase to about 300 Kp/s.

For maximum size packets, the SPC-1 can forward at about 220 M b/s. With word-swapping turned off, this increases to about 380 M b/s. This indicates that the SPC-2 should be able to do about 400 M b/s with the word-swapping code turned on, because of the faster instruction processing rate and
main memory bandwidth. It would not be likely to do much better than this, since at 400 M b/s throughput, it is using 80% of the PCI bus bandwidth (packets pass over the PCI bus twice).

3. The synchronous DRAM used in the FPX is organized into storage "chunks" of 16 words (128 bytes), because reading and writing short blocks is inefficient. Accounting for overheads in using the memory, each of the two DRAMs can support about 5 million such chunk accesses per second. Assuming worst-case packet sizes, how many chunk accesses per second are required to support link rates of 1 Gb/s? Also, assuming worst-case packet sizes, how much packet data can be stored in the DRAMs? How does this compare to the bandwidth-delay product, assuming a network round trip time of 100 MHz?

In the worst-case, the packets are just a little too big to fit in one chunk, meaning we need to do roughly twice as many memory accesses as we would in the ideal case. With 1 Gb/s links, this means we need about 4 Gb/s of memory bandwidth for the input link and another 4 Gb/s for the output link. In chunks per second, this is just under 4 million chunk accesses per second for each of the input and output link. Things can actually be worse than this, since the system is operated with a 2:1 speedup on the switch side, meaning that there can be time periods where the overall memory bandwidth usage peaks to 1.5 times the normal amount. This pushes the worst-case number of memory accesses to about 6 million per second for each of the input and output links.

Each of the two SDRAMs stores 64 M bytes. With the 2:1 inefficiency that arises from worst-case packet sizes, we have effectively a total of 64 M bytes. With 1 Gb/s links and an RTT of 100 ms, the bandwidth-delay product is about 12.5 M bytes, so the available memory is equivalent to about five times the bandwidth-delay product. This is a fairly typical number for a WAN router and is needed in the context of TCP flow control and simple FIFO queueing with tail discard. More intelligent queueing methods can significantly reduce the amount of memory needed.
4. For each of the networks shown below, given an expression for the topology using the series and parallel connection operations and the other topology construction operations. For each network topology that corresponds to a “standard” network type (e.g. a Benes network), also give the name and the appropriate symbol for the particular topology (e.g. \( B_{8,2} \)).
5. The digit reversal permutation \( R_{k,d} \) is the permutation obtained by replacing each number in the range from 0 to \( d^k - 1 \) with the number obtained by reversing the \( k \) digits in its base-\( d \) representation. So, for example, \( R_{4,2} \) maps the number 3 (binary representation 0011) to the number 12 (binary representation 1100). Show that the banyan network \( Y_{27,3} \) is strongly isomorphic to the network obtained by preceding the delta network \( D_{27,3} \) with the permutation \( R_{3,3} \). Do this by drawing pictures of the two networks and labeling them so as to demonstrate their relationship. Prove by induction that this property holds in general. That is, that \( Y_{n,d} \approx R_{k,d} \cdot D_{n,d} \) where \( k = \log_d n \).

The labeled banyan network is shown below.
The delta network preceded by the digit reversal permutation is shown below. The switches are labelled so as to highlight the strong isomorphism.

The general case can be proved by induction. Note that when \( k=1 \), the desired property holds because both networks are just single crossbars. When \( k>1 \),

\[
Y_{nd} = \tau_{nd,d} \cdot ((n/d) \cdot X_{d,d}) \cdot \tau_{d,nd,d} \cdot (d \cdot Y_{nd,d})
\]

\[
= \tau_{nd,d} \cdot ((n/d) \cdot X_{d,d}) \cdot \tau_{d,nd,d} \cdot (d \cdot (R_{k-1,d} \cdot D_{nd,d}))
\]

\[
= \tau_{nd,d} \cdot ((n/d) \cdot X_{d,d}) \cdot \tau_{d,nd,d} \cdot (d \cdot (R_{k-1,d} \cdot (d \cdot D_{nd,d})))
\]

\[
= \tau_{nd,d} \cdot ((n/d) \cdot X_{d,d}) \cdot (R_{k,d} \cdot (d \cdot D_{nd,d}))
\]

\[
= R_{k,d} \cdot (d \cdot D_{nd,d})
\]

The first line above is just an expanded version of the definition of the banyan network. In the second line, we've applied the induction hypothesis. The third line is just a regrouping. In the fourth line,
we've observed that since the perfect shuffle acts as a k digit right rotation, that the combination of this with a k-1 digit reversal is equivalent to a k digit reversal. The fifth line is based on the observation that \( R_{k,d;((n/d) \cdot X_{d,d})}; \tau_{a,n/d} \) is equivalent to its reversal. To see this, let \( x_{k-1}...x_1 \) be the base-d representation for a switch in the first stage of the network \( \tau_{a,n/d;((n/d) \cdot X_{d,d})};R_{k,d} \). The network inputs connected to this switch are

\[
0x_{k-1}...x_1, 1x_{k-1}...x_1, ..., d-1x_{k-1}...x_1
\]

and the network outputs connected to the switch are

\[
0x_1...x_{k-1}, 1x_1...x_{k-1}, ..., d-1x_1...x_{k-1}
\]

Now, let the switch numbers in \( \tau_{a,n/d;((n/d) \cdot X_{d,d})};\tau_{a,n/d} \) be \( y_{k-1}...y_1 \) where \( y_{k-1}...y_1 = x_1...x_{k-1} \). Note that switch \( y_{k-1}...y_1 \) is connected to network inputs

\[
0y_1...y_{k-1}, 1y_1...y_{k-1}, ..., d-1y_1...y_{k-1} = 0x_{k-1}...x_1, 1x_{k-1}...x_1, ..., d-1x_{k-1}...x_1
\]

and to network outputs

\[
0y_{k-1}...y_1, 1y_{k-1}...y_1, ..., d-1y_{k-1}...y_1 = 0x_{1}...x_{k-1}, 1x_{1}...x_{k-1}, ..., d-1x_{1}...x_{k-1}
\]

so, the two networks are equivalent, except for a renumbering of the switches.

The last line in the derivation above, follows from the definition of the delta network.

6. Find a set of routes that realizes the following permutation in the Benes network \( B_{16,2} \). Use the graph coloring approach to determine how to partition the paths among the subnetworks at each step. For each step in the routing process, show the connection graph, the coloring you found and how the connections were divided among the subnetworks.

<table>
<thead>
<tr>
<th>input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>14</td>
<td>4</td>
<td>11</td>
<td>15</td>
<td>2</td>
<td>13</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
The connection graph for the top level routing is shown at left, below, with edges colored appropriately. The corresponding top level routing is shown at right.

The connection graph for the top subnetwork is shown at left, with the routing at right. Below are the connection diagrams and routing for the top and bottom subnetworks of this network.
The connection graph for the top subnetwork is shown at left, with the routing at right. Below are the connection diagrams and routing for the top and bottom subnetworks of this network.

The overall routing is shown below.
7. (20 points) Let $d$ be a positive even integer and let $k$ and $h$ be positive integers with $k > h$.
Consider the network defined by the expression

$$N_{d,k,h} = ((d^{k-h}/2) \cdot D_{d,d}) \cdot \tau_{d^{k-h}/2} \cdot ((d^h/2) \cdot D_{d^{k-h},d}) \cdot \tau_{d^{k-h}/2,d} \cdot ((d^{k-h}/2) \cdot D_{d^{k-h},d})$$

Draw a picture of $N_{2,5,2}$.

How many paths are there between any input and output in this network? What about for the general case? How many inputs and outputs does this network have in the general case?

8 paths for the example network and $2d^h$ in the general case. The number of inputs and outputs is $d^h/2$.

Do a worst-case loading analysis of this network, similar to the one done on 4-37 and 4-38 of the notes. In particular, determine what speedup is needed for this network to be nonblocking in the sense defined on page 4-37 (express the required speedup as a function of the network parameters, $d$, $k$ and $h$).

The analysis is similar to the analysis given in the notes. If $\ell$ is a link in stage $i$ and $c = (x_i, y_i, \omega_i)$ is a session, then
Since a link in stage $i$ can be reached from at most $d_i$ inputs, the total traffic from the set of sessions whose cells can cross link $\ell$ (call this set $C_\ell$) is at most $\beta d_i$. That is,

$$\sum_{c_j \in C_\ell} \omega_j \leq \beta d_i$$

and similarly,

$$\sum_{c_j \in C_\ell} \omega_j \leq \beta d^{k+h-i}$$

Consequently, for any link $\ell$ in stage $i \leq h$,

$$\lambda_\ell(C) = \sum_{c_j \in C_\ell} \lambda_\ell(c_j) \leq d^{-i} \sum_{c_j \in C_\ell} \omega_j \leq \beta$$

and for any link $\ell$ in stage $i \geq k$,

$$\lambda_\ell(C) = \sum_{c_j \in C_\ell} \lambda_\ell(c_j) \leq d^{-(k+h-i)} \sum_{c_j \in C_\ell} \omega_j \leq \beta$$

and for any link $\ell$ in stage $i$ where $h < i < k$,

$$\lambda_\ell(C) = \sum_{c_j \in C_\ell} \lambda_\ell(c_j) \leq d^{-h/2} \sum_{c_j \in C_\ell} \omega_j \leq \beta d^{-h/2} 2 \min\{d^i, d^{k+h-i}\} \leq \beta d^{-(k+h)/2}/2 \leq \beta d^{[(k-h)/2]/2}$$

So, to avoid blocking, we need a speedup that satisfies $(1/\beta) \geq \max\{1, d^{[(k-h)/2]/2}\}$. For the example network, this means we need a speedup of 1.

8. Show how to modify the queueing analysis of the multistage switching network (starting on pages 4-40) to handle acknowledgement-based flow control, instead of grant-based flow control.

Most of the analysis does not need to change. The two parts that do change are the equations for $p_i$ and $q_i$, which are the probability distributions for the number of arriving cells and departing cells, respectively. If an upstream node has a cell to send to an SE in stage $i$, it will do so. Arriving cells are accepted, so long as there is room for them. So $p_i(j,s) = B(j,d,a_i) = \sum_{j=0}^{d} \binom{d}{j} a_i^j (1-a_i)^{d-j}$ where $a_i$ is the probability that the upstream SE has a cell to send to the stage $i$ SE and is defined by the same equation as in the original analysis.
A cell leaves a stage $i$ SE if the downstream node acknowledges it. Consequently, $q_i(j,c) = B(j,c,b_i)$ where $b_i$ is now the probability of receiving an acknowledgement and is equal to

$$b_i = \sum_{s,c} \pi_{i+1}(s,c) \sum_{h=0}^{d-1} \left( \frac{d-1}{h} \right) a_{i+1}^b (1-a_{i+1})^{d-1-h} \min\{1, (B-s)/(h+1)\}$$

9. (50 points) On the web site, you will find a program for simulating a buffered multistage switch, using a Benes network topology and dynamic routing. As written, the program implements shared buffer switch elements in which there is no limit on the number of buffer slots that can be used by the cells going to a single output. This can result in low throughput in situations where outputs are overloaded for extended periods of time.

Modify the traffic source model so that it causes heavy loads at each of the outputs during successive time periods equal to $P$ of the switch's internal cell times. Specifically if $t$ is the current simulation time and $n$ is the number of ports, then whenever $(t/p) \mod n = i$, make the load at output $i$ four times larger than the specified average load. Simulate a 64 port switch with a speedup of 1 and 8 port switch elements having 128 buffer slots each. Use this to produce curves of input load vs. output load (similar to the one on the left side of page 4-45) for $P = 5, 10, 20, 40$ and $80$. Let the input buffer size be 64, let the resequencer size be 64 with an age threshold of 60 and let the output buffer size be 256.

The modified portions of the Srcsnk.c file are shown below, with changes highlighted.

```c
#include "netsim.h"
#include "Srcsnk2.h"

Srcsnk::Srcsnk(register int n1, double load1, double speedup1, int period1) {
    // Initialize source/sink.
    n = n1; load = load1; speedup = speedup1; period = period1;
    lo = 0; hi = lo + n/speedup; if (hi >= n) hi -= n;
    if (load <= 0 || load > 1) fatal("illegal load");
    clearstats();
}

void Srcsnk::gen(Flagvec gv, Cvec outp) {
    // Generate packets into outp.
    int h, j, target;
    target = (gtime/period) % n;
    for (j = 0; j < n; j++) {
        if (((lo < hi && (lo <= j && j < hi)) || (lo >= hi && (j < hi || j >= lo))) &&
            randfrac() < load) {
            nout[j]++;
            if (gv[j]) {
                if ((outp[j] = new Cell) == 0)
                    fatal("Out of space");
                outp[j]->scr3 = gtime;
                h = randint(0,n-1);
                if (randint(0,n-1) < 3)
                    h = target;
                outp[j]->adr = h;
            }
        }
    }
}
```
The chart obtained from running the simulation is shown below. As the period is increased, the network throughput drops, as one would expect.

Modify the code implementing the switch element so that it reserves 4 buffer slots for each output (remaining buffer slots are shared and can be used for any output). In this modified switch element, grants are given to all upstream neighbors, every cell time, but arriving cells are discarded if there is no room for them, either in the reserved space for the target output, or in the remaining shared space. Re-run your simulation using the modified switch elements. Compare your results and comment on the differences observed.

The modifications to Network.h and Network.c are shown below, with modifications highlighted.

```c
// Common declarations for a Network class
const int Maxd=32;  // Maximum node size
const int Maxk=12;  // Maximum log_d(n)
const int Maxm=1024;  // Maximum number of nodes per stage

struct Ic {
    Flag gu;  // upstream grant signal
    int nrcvd;  // number received on this input
    int nidle;  // number of times input has grant but no cell
    int nblk;  // number of times input has no grant
};

struct Oc {
    int nq;  // number queued for this output
    Cell *op;  // cell going to next stage
    Flag gd;  // downstream grant
};

struct Node {
    int intie;  // input tie breaker variable

    lo = hi; hi = lo + n/speedup; if (hi >= n) hi -= n;

    int lo = hi; hi = lo + n/speedup; if (hi >= n) hi -= n;
}
```

Modify the code implementing the switch element so that it reserves 4 buffer slots for each output (remaining buffer slots are shared and can be used for any output). In this modified switch element, grants are given to all upstream neighbors, every cell time, but arriving cells are discarded if there is no room for them, either in the reserved space for the target output, or in the remaining shared space. Re-run your simulation using the modified switch elements. Compare your results and comment on the differences observed.

The modifications to Network.h and Network.c are shown below, with modifications highlighted.

```c
// Common declarations for a Network class
const int Maxd=32;  // Maximum node size
const int Maxk=12;  // Maximum log_d(n)
const int Maxm=1024;  // Maximum number of nodes per stage

struct Ic {
    Flag gu;  // upstream grant signal
    int nrcvd;  // number received on this input
    int nidle;  // number of times input has grant but no cell
    int nblk;  // number of times input has no grant
};

struct Oc {
    int nq;  // number queued for this output
    Cell *op;  // cell going to next stage
    Flag gd;  // downstream grant
};

struct Node {
    int intie;  // input tie breaker variable
```
int outtie; // output tie breaker variable
Cqueue sq; // shared cell queue
Ic *ic; // array of input circuits
Oc *oc; // array of output circuits
};

class Network {
  int d; // # of inputs per node
  int k; // log_d(n)
  int n; // # of inputs to network (d**k)
  int ld; // log_2(d)
  int m; // # of nodes/stage (n/d)
  int sqsiz; // size of shared buffer
  Node* node[2*Maxk]; // two dimensional array of nodes
  int delay[Maxn]; // delay[j] = sum of delays of cells
                   // exiting from output j
  int nout[Maxn]; // number received at each output
  int nidle[Maxn]; // times each output was idle
  int nbkl[Maxn]; // times each output was blocked
  int stattime; // time when stats were last cleared
  int *next[2*Maxk]; // next[i][j] is end of link i,j in stage i+1
  int *prev[2*Maxk]; // prev[i][j] is end of link i,j in stage i-1

  // private routines
  void pre_grant(Flagvec);
  void node_grant(int,int);
  void pass_grant(int,int);
  void post_grant(Flagvec);
  void pre_flow(Cvec);
  void node_flow(int,int);
  void pass_flow(int,int);
  void post_flow(Cvec);
  void copy_flow(Node*,Cell*,int);

  int suc(int i, int j) {return i < k ? rho(j,k+1-i) : irho(j,i+2-k);}
  int pred(int i, int j) {return i <= k ? irho(j,k+2-i) : rho(j,i+1-k);}
  int rho(int,int);
  int irho(int,int);

  public:
    Network(int d, int k, int sqsiz) {
      // Initialization for a network with nodes of size d and n1 stages.
      register int i,j,h;
      d = d1; k = k1; sqsiz = sqsiz1;
      m = (int) pow(double(d), double(k-1));
      n = m*d;
      if (d < 2 || d > Maxd || k < 1 || k > Maxk || m < 1 || m > Maxm
          || n < 1 || n > Maxn)
        fatal("Network: constructor parameter out of range");
      for (i = d, ld = 0; i > 1; i >>= 1) {
        if (i & 1) fatal("Network: d must be power of 2");
      }
    }

    void grant(Flagvec,Flagvec);
    // propagate grants from outputs to inputs
    void flow(Cvec,Cvec);
    // push cells from inputs to outputs
    void stats(int); // print statistics about network
    void clearstats(); // clear out statistics
    void dump(); // dump network status
};

#include "netsim.h"
#include "Cqueue.h"
#include "Network2.h"

Network::Network(int d1, int k1, int sqsiz1) {
  // Initialization for a network with nodes of size d1 and n1 stages.
  register int i,j,h;
  d = d1; k = k1; sqsiz = sqsiz1;
  m = (int) pow(double(d), double(k-1));
  n = m*d;
  if (d < 2 || d > Maxd || k < 1 || k > Maxk || m < 1 || m > Maxm
      || n < 1 || n > Maxn)
    fatal("Network: constructor parameter out of range");
  for (i = d, ld = 0; i > 1; i >>= 1) {
    if (i & 1) fatal("Network: d must be power of 2");
  }
ld++;  
}
for (i = 1; i <= 2^k-1; i++) {  
    node[i] = new Node[m];
    next[i] = new int[n];
    prev[i] = new int[n];
    for (j = 0; j < n; j++) {  
        next[i][j] = suc(i,j);  
        prev[i][j] = pred(i,j);
    }
    for (j = 0; j < m; j++) {  
        node[i][j].intie = randint(0,d-1);  
        node[i][j].outtie = randint(0,d-1);  
        node[i][j].ic = new Ic[d];  
        node[i][j].oc = new Oc[d];  
        for (h = 0; h < d; h++) {  
            node[i][j].ic[h].gu = 0;  
        }
    }
    clearstats();
}
void Network::node_grant(register int i, register int j) {  
    // Do grant propagation for node i,j.  
    // Always issue grants on all inputs.  
    register int h;  
    register Node *p = &(node[i][j]);  
    for (h = 0; h < d; h++) p->ic[h].gu = 1;
}
void Network::pre_flow(Cvec inp)  
// Preprocessing for flow.  
{  
    register int j,h;  
    register Cell *p;
    for (j = 0; j < m; j++) {  
        for (h = 0; h < d; h++) {  
            p = inp[j*d+h]; inp[j*d+h] = NULL;  
            if (p != NULL && node[1][j].sq.qsiz() < sqsiz) {  
                p->scr2 = gtime;  
                node[1][j].sq.enq(p);  
                node[1][j].ic[h].nrcvd++;  
                if (node[1][j].ic[h].gu == 0)  
                    fatal("pre_flow logic error");  
            } else if (p != NULL) {  
                delete p;  
            } else if (node[1][j].ic[h].gu)  
                node[1][j].ic[h].nidle++;
            else  
                node[1][j].ic[h].nblk++;
        }
    }
}
void Network::node_flow(register int i, register int j) {  
    // Flow cells through node i,j.  
    register int h, slot;
    register Node* p = &(node[i][j]);  
    for (h = 0; h < d; h++) {  
        p->ic[h].gu = 1;  
    }  
    for (j = 0; j < m; j++) {  
        for (h = 0; h < d; h++) {  
            p = node[i][j].ic[h].gu;  
            if (p != NULL) {  
                p->scr2 = gtime;  
                node[1][j].sq.enq(p);  
                node[1][j].ic[h].nrcvd++;  
                if (node[1][j].ic[h].gu == 0)  
                    fatal("node_flow logic error");  
            } else if (p != NULL) {  
                delete p;  
            } else if (node[1][j].ic[h].gu)  
                node[1][j].ic[h].nidle++;
            else  
                node[1][j].ic[h].nblk++;  
        }
    }
}
register Cell *x, *y;

slot = 0;
for (x = p->sq.qfirst(); x != NULL; x = y) {
    y = p->sq.qnext(x);
    if (i >= k) {
        h = (x->adr >> ld*((2*k-1)-i)) & ((1 << ld) - 1);
        if (p->oc[h].op == NULL && p->oc[h].gd) {
            p->oc[h].nq--;
            p->oc[h].op = x;
            p->sq.qdel(x);
        }
    } else {
        h = (p->outtie + slot++) % d;
        if (p->oc[h].op == NULL && p->oc[h].gd) {
            p->oc[h].op = x;
            p->sq.qdel(x);
        }
    }
} 

p->outtie++;
if (p->sq.qsiz() > sqsiz)
    fatal("node_flow:: too many cells in queue");
}

void Network::pass_flow(register int i, register int j) {
    // Pass cells from node i,j OCs to downstream neighbors
    // If passed cells don't fit in the available buffer space,
    // then discard them.
    int h, j1, h1, r, s, t;
    Node* p = &(node[i][j]);
    Node* q;

    for (h = 0; h < d; h++) {
        j1 = next[i][j*d+h] / d;
        h1 = next[i][j*d+h] % d;
        q = &(node[i+1][j1]);
        if (p->oc[h].op != NULL) {
            s = (p->oc[h].op->adr >> ld*((2*k-1)-(i+1)))
                & ((1 << ld) - 1);
            t = 0;
            for (r = 0; r < d; r++)
                t += max(0, q->oc[r].nq - 4);
            if (i < k && q->sq.qsiz() < sqsiz) {
                q->ic[h1].nrcvd++;
                q->sq.enq(p->oc[h].op);
            } else if (i >= k &&
                       (q->oc[s].nq <= 4 || t < sqsiz - 4*d)) {
                q->oc[s].nq++;
                q->ic[h1].nrcvd++;
                q->sq.enq(p->oc[h].op);
            } else {
                delete p->oc[h].op;
            }
        } else if (q->ic[h1].gu) {
            q->ic[h1].nidle++;
        } else {
            q->ic[h1].nblk++;
        }
    }
}
The chart obtained using the modified simulation is shown below. There are two differences worth noting. First, the maximum output loads are significantly higher when the periods are long. This improvement results from the reserved buffer slots, which prevent any single output from hogging all the available slots, and preventing traffic from reaching other outputs. The second difference is that at lower input loads, we get lower output load. This occurs because in this version cell discarding can occur at the switch elements, while in the earlier version cell discarding was allowed to occur only at the input queue.