February 19, 1998

SRC Research Report 150

Smooth Scheduling in a Cell-Based Switching Network

Thomas L. Rodeheffer
and
James B. Saxe

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

http://www.research.digital.com/SRC/

Smooth Scheduling in a Cell-Based Switching Network

Thomas L. Rodeheffer and James B. Saxe

February 19, 1998

Abstract

This report describes a method for scheduling data cell traffic through a crossbar switch in such a way as to guarantee that the time slots for each flow, and optionally for certain aggregates of flows, are approximately uniformly distributed throughout the duration of the scheduling frame. Such a "smooth" schedule achieves reductions both in the latency of data flows and in the need for associated memory buffers. Our method, called recursively balanced scheduling, can be used to compute smooth schedules even under conditions of maximum load, where the bandwidth requirements of the flows to be scheduled are sufficient to consume the entire capacity of the switch. This method has been implemented for a working high-speed ATM switching network, AN2. We describe the implementation, its performance, and possible future improvements.

Contents

1. Introduction

This report describes a method for scheduling data cell traffic through a crossbar switch so that the cells for each flow are forwarded with little variation in latency. The method takes a set of *bandwidth requests* and prepares a schedule for a sequence of time slots called a *scheduling frame*. The schedule specifies which flows may forward cells in which time slots. This schedule is then applied to actual cell traffic, repeating over and over until a change in the set of bandwidth requests triggers the preparation of a new schedule. We view preparation of the schedule as a policy and application of the schedule as a mechanism. Typically software will implement the policy and hardware the mechanism. The property of having little variation in latency we informally call *smoothness*; schedules whose flows are smooth we call *smooth schedules*. We call our method for preparing smooth schedules *recursively balanced scheduling*. It works by recursively halving the scheduling frame while splitting the bandwidth requests as evenly as possible.

We assume a simple crossbar switch that can accept only one cell on each input port and emit only one cell on each output port during each time slot. Hence the scheduling plan must not contain any time slots with *conflicts*, in which the same input or output would be used more than once. This requirement creates interactions between flows and makes it difficult to construct a smooth schedule.

Any scheduling plan that satisfies the bandwidth requests will provide each flow's requested bandwidth over the duration of the scheduling frame. However, if a flow's time slots are clustered together, that flow will experience a large variation in latency. In the worst case the variation in latency can approach the duration of the frame and the buffering required just to cover this variation in latency can approach the flow's bandwidth per frame. A smooth schedule reduces both the latency of data flows and the need for associated memory buffers.

The recursively balanced scheduling algorithm can be used to compute smooth schedules even under conditions of maximum load, where the bandwidth requirements of the flows to be scheduled are sufficient to consume the entire capacity of the switch. The method described here has been implemented for a working high-speed ATM switching network, AN2, which has been in service at our laboratory since 1994. We describe the implementation, its performance, and possible future improvements.

2. Background

A *crossbar switch* is a device that has some number of *input ports*, some number of *output ports*, and a connection mechanism that can be dynamically *configured* to connect any input to any output, allowing simultaneous transmission of different data across different input-output connections. Only one input may be connected to any given output at any given time. Data destined for the given output from other inputs must either be discarded or be *buffered* at the inputs. We concern ourselves here with input-buffered crossbar switches, rather than other types of switches, such as those that use shared memory, a shared bus, or buffering in the switch fabric [11, 18].

Data travelling through the switch is organized into *flows*, each of which has fixed input and output ports, although there may be several flows with the same input and output ports, as happens for example with virtual circuits. It is often desirable to provide *guarantees of service* to particular flows. One discipline for providing such guarantees is as follows:

- Organize each flow into *cells* of a fixed size. A common example is the ATM standard 53-byte cell, consisting of 48 payload bytes and 5 header bytes.
- Divide time at the switch into fixed *slots*. During each slot the switch can be configured to implement a different connection of inputs to outputs, and each input can send at most one cell.
- Group time slots into equal-sized batches of consecutive time slots. These batches are called *scheduling frames*.
- Compute a *schedule* associating each slot position t within a frame with a set of nonconflicting flows that will be given absolute priority for use of the switch during the t-th slot of each frame. Two flows *conflict* if they have any port in common.

Under this discipline, a scheduled flow receives a bandwidth guarantee of m cells per frame, where m is the number of slots in the schedule that contain the flow. This type of service is commonly called *constant bit rate* (CBR) service.

Input and output ports that are not used by any scheduled flow during a given slot might be used for nonscheduled, *opportunistic* flows. Such flows would receive no bandwidth guarantee. This type of service is commonly called *available bit rate* (ABR) service. Although this report is not concerned with ABR service, the manner in which the CBR schedule is computed can affect the bandwidth available for ABR. This effect will be discussed briefly in Section 6.

Figure 1 illustrates a crossbar switch. During each time slot the input/output port matcher evaluates the schedule and the set of flows that have cells available and decides which flows will send a cell through the crossbar in the next time slot.

Flows may be *unicast* flows, which must use exactly one output port of the switch, or *multicast* flows, which can use more than one output port. (In either case, the flow must use exactly one input port.) For the present, we consider only unicast flows. A rudimentary way to handle multicast flows will be discussed in Section 7.

Suppose we have a scheduling frame size of n slots and a set B of bandwidth requests, where each request has the form "flow f needs m cells per frame of bandwidth from input port i to output port o." Given this set B of bandwidth requests, the schedule must satisfy the following two constraints to provide all the requested guarantees of service:

- Each flow must be scheduled into the requested number of slots.
- Two flows that *conflict* with each other by using the same input port or the same output port must not be scheduled into the same time slot.

It is easy to see that a necessary condition for the existence of such a *legal* schedule is that for each input or output port p, the total bandwidth requested by all flows using port p (called the *load* on p) must be no greater than the bandwidth of port p, that is, n cells per frame. It turns out that this is also a sufficient condition: if no input or output port has a load of more than n cells per frame, then a legal schedule is possible [10].

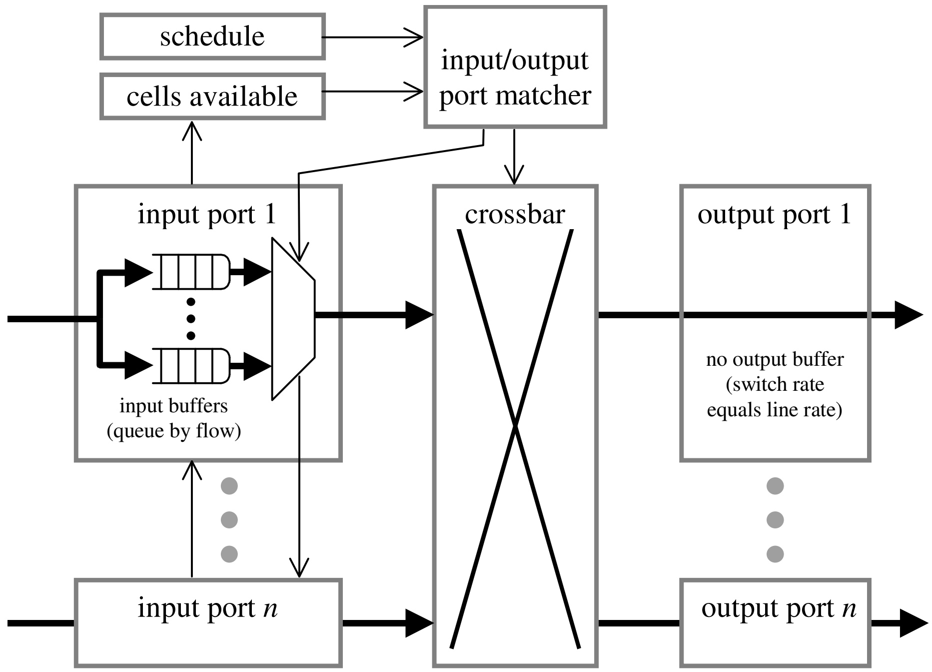

Figure 1: A crossbar switch.

Usually, if any legal schedule exists, there are many such schedules. These schedules differ in the timing of the forwarding of cells, and, as a consequence, some of these schedules require more buffering in the switch than others require. We are interested in finding schedules that provide guarantees that allow us to build a switch with less buffering than would otherwise be required. In particular, we are interested in "smooth" schedules that distribute the scheduled slots so that cells of an input port, an output port, or a flow are processed at a relatively steady rate throughout the scheduling frame. Such schedules also result in lower worst-case latency than arbitrary legal schedules.

For example, suppose that the data cells of a flow arrive at the switch at a relatively steady rate, but pass through the switch according to a legal schedule in which all the scheduled slots for the flow are clustered at the end of the frame. The latency period during which cells of the flow must wait in the input buffers could be quite long—almost a frame time—and the amount of buffer space used by cells in the flow could be quite large—almost the per-frame bandwidth of the flow.

Another problem related to clustering occurs in applications where the outputs of a switch are used to drive *communication links* that run at bandwidths lower than the per-port bandwidth of the switch. For example, an output of a high-speed switch might be used to drive several lower-speed links, with data cells being demultiplexed onto the appropriate links as they arrive at the switch outputs. In such cases, *output buffering* is needed to hold the cells that arrive (at high speed) at the output port until they can be transmitted (at lower speed) over the appropriate link. Figure 2 illustrates this type of switch. Suppose the time slots reserved for data flows using a given output link are clustered into one part of the schedule. Then the amount of output buffering needed at each output port could approach the per-frame bandwidth of the link, and the latency introduced by output buffering could approach the frame time.

The recursively balanced scheduling method we describe below avoids the problems associated with clustering by distributing the time slots for each flow approximately uniformly throughout the scheduling frame. The method can also ensure that the time slots during which any given output link is used are distributed approximately uniformly throughout the scheduling frame. The method can compute a schedule having both these smoothness properties regardless of the complexity of the pattern of bandwidth requests and regardless of the amount of load, provided there is no input or output port whose load exceeds the switch's per-port bandwidth and no output link whose load exceeds the bandwidth of that link. Previously known methods for regulating switch traffic cannot simultaneously handle high loads and guarantee smoothness [3, 7, 10].

The problem of computing a schedule from a set of bandwidth requests also arises in the context of time-domain multiplexed access (TDMA) networks such as satellite communications systems. However, such systems take an appreciable time to reconfigure the switch, so work in this area [8, 13, 17] has concentrated on minimizing the length of the schedule and the number of distinct configurations needed rather than on smoothness. Our model assumes that reconfiguration occurs every time slot, pipelined with the data transfer, and therefore is effectively instantaneous.

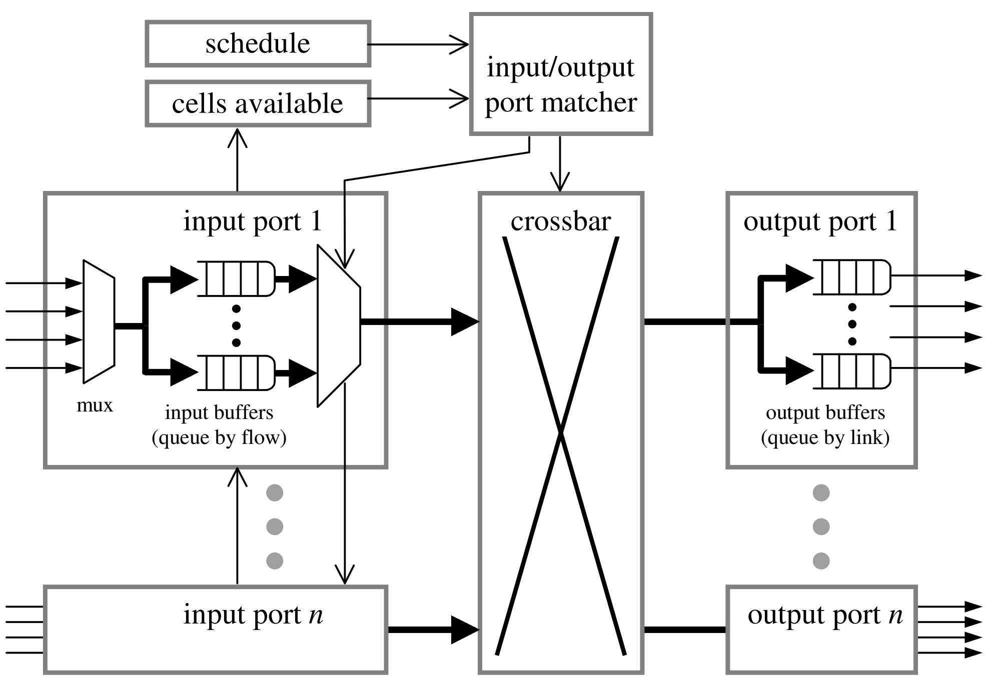

Figure 2: Crossbar switch with several lower-speed links per port.

3. The basic recursively balanced scheduling method

The input to the basic recursively balanced scheduling method consists of

- the frame size n, which must be an exact power of two, and
- a set B of bandwidth requests. Each bandwidth request b has four fields, $b.f$, $b.i$, $b.o$, and $b.m$, where
  - $b.f$ is a flow identifier, unique within B,
  - $b.i$ is the input port for b, one of in1, in2, etc.,
  - $b.o$ is the output port for b, one of out1, out2, etc., and
  - $b.m$ is the bandwidth for b, given as a non-negative integer number of cells per frame.

In Section 6 we will present modifications to the method to handle arbitrary-sized frames.

Because flow identifiers are unique within B, it is unambiguous to refer to the unique bandwidth request b for flow f in B as simply $B[f]$. The sum of the bandwidths for all bandwidth requests in B we call the *load* of B. We define the *load* in B for a flow f, an input port i, or an output port o, as the sum of the bandwidths for all bandwidth requests with that flow, input port, or output port. In the case of a flow, there will be at most one such bandwidth request: $B[f]$ if it exists. Table 1 summarizes the definitions.

$load(B)$	$\equiv$	$\sum_{b\in B} b.m$	load of B
$load(B,f)$	$\equiv$	$\sum_{b\in B:b.f=f} b.m$	load in B due to flow f
$load(B,i)$	$\equiv$	$\sum_{b\in B:b.i=i} b.m$	load in B on input port i
$load(B,o)$	$\equiv$	$\sum_{b\in B:b.o=o} b.m$	load in B on output port o

Table 1: Load definitions.

The bandwidth request set B is *feasible for frame size n* if the load in B on each input and output port is at most n. If the bandwidth request set is not feasible, no schedule is possible. The scheduling method described below works for any feasible request set.

The output is a schedule S associating each time slot t in the range $1,\ldots,n$ with a set $S[t]$ of flows. The schedule S has the following properties:

LEGAL.1. For any flow f, the number of time slots in which flow f appears in schedule S equals the load in B due to f. Formally,

$$\left|\{t : f \in S[t]\}\right| = load(B,f).$$

LEGAL.2. No two flows that use the same input or output port appear in the same time slot of S.

S is thus a legal schedule for the request set B. The schedule S also has certain smoothness properties discussed in more detail below.

3.1. Recursive splitting of the scheduling frame

The recursively balanced scheduling method works by recursive splitting of the scheduling frame. Because the frame size is a power of two, each split divides the scheduling frame exactly in half.

The base case of the recursion is the case $n = 2^0 = 1$; that is, there is just one slot in the frame. Since the bandwidth request set B is required to be feasible for the frame size $n = 1$, there can be at most one request in B for a flow of non-zero bandwidth for any given input or output port, and each such request must be for a bandwidth of 1. So the method simply returns a one-slot schedule S, where $S[1]$ is the set of all positive-bandwidth (i.e., bandwidth 1) flows in B.

The other case is that $n = 2^{k+1}$ for some non-negative integer k. In this case, the method *splits* the bandwidth request set B into two bandwidth request sets $B1$ and $B2$ having the following properties:

SPLIT.1. Each flow f has the same input and output ports in $B1$ and $B2$ as in B. Formally,

if $B1[f]$ exists then $B1[f].i = B[f].i$ and $B1[f].o = B[f].o$, and
if $B2[f]$ exists then $B2[f].i = B[f].i$ and $B2[f].o = B[f].o$.

SPLIT.2. Each flow f has its load in B divided between $B1$ and $B2$. Formally,

$$load(B, f) = load(B1, f) + load(B2, f).$$

SPLIT.3. Each flow f has its load divided as equally as possible. Formally,

$$\left| load(B1, f) - load(B2, f) \right| \le 1.$$

SPLIT.4. Each input port i has its load divided as equally as possible. Formally,

$$\left| load(B1,i) - load(B2,i) \right| \le 1.$$

SPLIT.5. Each output port o has its load divided as equally as possible. Formally,

$$\left| load(B1,o) - load(B2,o) \right| \le 1.$$

The method then recursively computes schedules $S1$ for $B1$ and $S2$ for $B2$, each with frame size $n/2 = 2^k$, and concatenates those schedules to produce the schedule S for B.

In order for the splitting step to be correct, $B1$ and $B2$ must each be feasible for frame size $n/2 = 2^k$. Properties SPLIT.1 and SPLIT.2 imply that the port loads in $B1$ and $B2$ sum to the corresponding port loads in B. Since B is feasible for frame size n, no port load in B can exceed n. Adding properties SPLIT.4 and SPLIT.5, along with the fact that n is even, allows the conclusion that no port load in $B1$ or $B2$ will exceed $n/2$. Hence $B1$ and $B2$ will indeed each be feasible for frame size $n/2$.

We must also verify that the concatenation of $S1$ and $S2$ produces a legal n-slot schedule S that satisfies all the bandwidth requests in B. This is guaranteed by properties SPLIT.1 and SPLIT.2.

Property SPLIT.3 leads to a per-flow smoothness property that will be discussed later. Properties SPLIT.4 and SPLIT.5, in addition to guaranteeing that the subproblems are feasible, also lead to a per-port smoothness property.

3.2. The splitting step

The splitting step can be accomplished as follows. For each bandwidth request *b* in *B*, let *b1* and *b2* be bandwidth requests (called *subrequests* of *b*) each with the same flow, input port, and output port as *b* and as nearly as possible half the bandwidth. See Table 2.

$b1.f \equiv b.f$	$b2.f \equiv b.f$
$b1.i \equiv b.i$	$b2.i \equiv b.i$
$b1.o \equiv b.o$	$b2.o \equiv b.o$
$b1.m \equiv \lfloor b.m / 2 \rfloor$	$b2.m \equiv \lceil b.m / 2 \rceil$

Table 2: Subrequest definition.

When the bandwidth of *b* is odd, subrequest *b1* gets half the bandwidth rounded down to an integer and *b2* gets half the bandwidth rounded up. Note that $b1.m + b2.m = b.m$ and $0 \le b2.m - b1.m \le 1$.

For each bandwidth request *b* in *B*, either *b1* or *b2* will be allocated to *B1* and the other subrequest will be allocated to *B2*. Any such allocation of the subrequests to *B1* and *B2* will make *B1* and *B2* satisfy properties SPLIT.1-3. It remains to describe how to allocate the subrequests so that properties SPLIT.4 and SPLIT.5 are satisfied.

For each even-bandwidth request *b* in *B*, *b1* and *b2* are equal, so the allocation is arbitrary. We will ignore even-bandwidth requests and will consider only odd-bandwidth requests for the remainder of this section.

For an odd-bandwidth request *b*, subrequest *b2* has bandwidth greater by one cell per frame than *b1*. We say that *b favors* whichever subrequest set receives subrequest *b2*, the subrequest of greater bandwidth. In order to satisfy properties SPLIT.4 and SPLIT.5, we must guarantee that the following properties hold:

ALLOC.1. For each input port *i*, the number of odd-bandwidth requests for *i* favoring *B1* and the number favoring *B2* differ by at most one.

ALLOC.2. For each output port *o*, the number of odd-bandwidth requests for *o* favoring *B1* and the number favoring *B2* differ by at most one.

We will achieve these properties by reducing the problem to that of finding a 2-coloring of a certain graph.

First, let us consider property ALLOC.1. We achieve this property by forming pairs of odd-bandwidth requests that have the same input port and then allocating subrequests so that the members of a pair favor different subrequest sets. The *input pairing*, *IP*, is a set of unordered pairs of odd-bandwidth requests. We call each element of *IP* an *input pair*. *IP* must satisfy the following properties:

INPAIR.1. Each request appears at most once (as a member of an input pair) in *IP*, and no request can be paired with itself.

INPAIR.2. If $\{b, c\} \in IP$, then $b.i = c.i$.

INPAIR.3. For any input port *i*, there is at most one odd-bandwidth request for *i* that does not appear (as a member of an input pair) in *IP*.

To construct *IP* we simply pair off as-yet-unpaired same-input odd-bandwidth requests until there is at most one unpaired odd-bandwidth request left for each input port. Subsequently we will allocate subrequests so that the members of each input pair favor different subrequest sets. In such an allocation, the load on any input port due to any input pair will divide exactly evenly and we say that the input pair is *balanced*. Since at most one request for input port *i* appears in no input pair, the load on *i* will be divided as equally as possible.

Similarly, we achieve property ALLOC.2 by forming pairs of odd-bandwidth requests that have the same output port and then allocating subrequests so that the members of a pair favor different subrequest sets. The *output pairing*, *OP*, is a set of unordered *output pairs* of odd-bandwidth requests. *OP* must satisfy the following properties:

OUTPAIR.1. Each request appears at most once in *OP*, and no request can be paired with itself.

OUTPAIR.2. If $\{b, c\} \in OP$, then $b.o = c.o$.

OUTPAIR.3. For any output port *o*, there is at most one odd-bandwidth request for *o* that does not appear in *OP*.

We construct *OP* in an analogous way to *IP*.

To achieve both ALLOC.1 and ALLOC.2 we must allocate the subrequests so that all input pairs and output pairs are balanced simultaneously. As we will show below, this allocation problem is equivalent to finding a 2-coloring of a graph *G* whose vertex set is the set of odd-bandwidth requests in *B* and whose edge set is the (labeled) union of *IP* and *OP*. We need a labeled union because it is possible for $\{b, c\} \in IP$ and $\{b, c\} \in OP$, in which case there must be two edges between *b* and *c*.

Observe that each vertex *b* in *G* has degree at most two: one incident edge corresponding to *b*'s membership in an input pair (if any), and one incident edge corresponding to *b*'s membership in an output pair (if any). Suppose we have a 2-coloring of *G* using the colors white and black. For each white odd-bandwidth request, allocate the subrequests to favor *B1*, and for each black odd-bandwidth request, allocate the subrequests to favor *B2*. Now consider any input pair $\{b, c\} \in IP$. Since $\{b, c\}$ is an edge in *G*, *b* and *c* must have opposite colors and hence the input pair is balanced. The same reasoning applies to any output pair. Hence all input pairs and output pairs are balanced and we achieve both ALLOC.1 and ALLOC.2.

Finding a 2-coloring of *G* is easy. Since each vertex has degree at most two, *G* consists of a disjoint collection of paths and cycles. Furthermore, since a vertex with two incident edges must have one edge in *IP* and one in *OP*, the total number of edges (and hence vertices) around any cycle must be even. So to produce a 2-coloring, we simply traverse each path and cycle in turn, assigning alternating colors to the vertices. An example of such a coloring appears in Figure 3 in the next section.

3.3. An example of the splitting step

Here is an example to illustrate the splitting step. Suppose that the frame size is 32 and we have the bandwidth request set *B* shown in Table 3. Checking the port loads, as shown in Table 4, we see that no input or output port load exceeds 32, and thus the bandwidth request set is feasible.

We identify the bandwidth requests by their unique flow ids. The easy part of splitting *B* into *B1* and *B2* is to deal with the even-bandwidth requests, which split exactly in half. The even-bandwidth requests are f4 and f11. Request f4 splits into two subrequests of bandwidth 3 and request f11 into two subrequests of bandwidth 1. In the final result shown in Table 5, shaded horizontal lines mark these even-bandwidth requests.

flow id	input port	output port	bandwidth
f1	in1	out1	9
f2	in1	out2	1
f3	in1	out3	11
f4	in1	out4	6
f5	in2	out1	5
f6	in2	out2	15
f7	in2	out3	11
f8	in3	out1	13
f9	in3	out1	5
f10	in3	out3	9
f11	in3	out4	2
f12	in3	out4	3
f13	in4	out4	15
f14	in4	out4	3

Table 3: Bandwidth request set *B*.

port	load
in1	27
in2	31
in3	32
in4	18
out1	32
out2	16
out3	31
out4	29

Table 4: Port loads in *B*.

The hard part is to deal with the odd-bandwidth requests, namely f1–f3, f5–f10, and f12–f14. Each such request will split into two subrequests whose bandwidths differ by one cell per frame. For example, the request f1 has bandwidth 9; its subrequests will have bandwidths 4 and 5. We must decide which requests favor *B1* and which favor *B2*.

We begin by constructing the input and output pairings. Let us consider the input pairing first. We begin by noticing that f1 and f2 use the same input port, in1, so they can be paired. The only remaining flow using in1 is f3, so it must go unpaired. Similarly we can pair f5 with f6 leaving f7 unpaired. Input port in3 appears in four odd-bandwidth requests, which we group into two pairs, {f8, f9} and {f10, f12}. Finally the two requests f13 and f14 that use in4 are paired with each other. The final input pairing is

$$IP = \{\{f1, f2\}, \{f5, f6\}, \{f8, f9\}, \{f10, f12\}, \{f13, f14\}\}.$$

In the same way, we can construct an output pairing, such as

$$OP = \{\{f1, f5\}, \{f8, f9\}, \{f2, f6\}, \{f7, f10\}, \{f12, f13\}\}.$$

For example, f7 and f10 can be paired since they share output port out3. The only remaining flow requesting out3 is f3, so it must remain unpaired. Figure 3 shows the paths and cycles produced by this input and output pairing. To help illustrate the fact that cycles must be even, edges resulting from the input pairing *IP* are drawn with thin lines and edges resulting from the output pairing *OP* are drawn with thick lines.

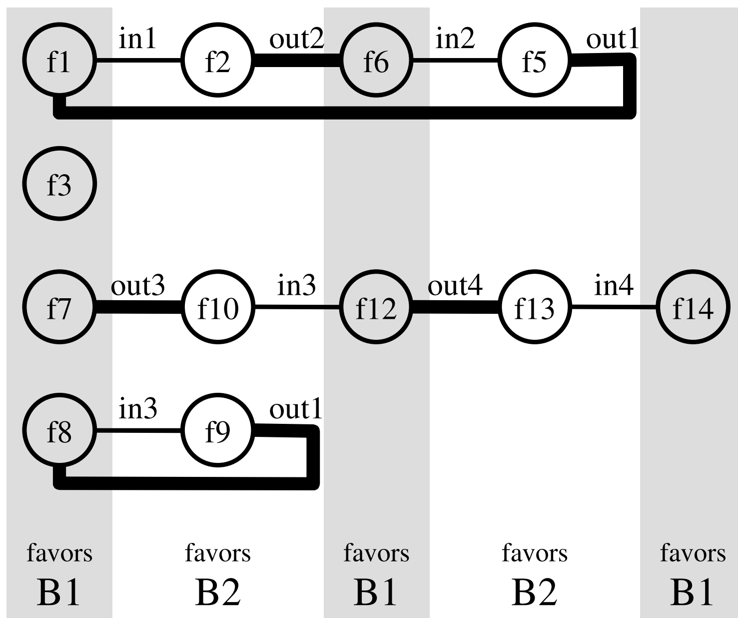

Figure 3: Paths and cycles of odd-bandwidth requests.

Next we traverse the paths and cycles, alternately favoring *B1* and *B2*. The columns in Figure 3 are shaded to show this alternation. Note that we have some freedom in whether an odd-length path on balance favors *B1* or *B2*. If we wanted to ensure that the total load of *B* was divided as evenly as possible (which is not a requirement), we could do so by pairing up odd-length paths.

Combining the subrequests of the odd-bandwidth requests with the subrequests of the even-bandwidth requests already discussed, we arrive at the subrequest sets *B1* and *B2* shown in Table 5.

Table 6 shows the input and output port loads in *B1* and *B2*. Note that the load on each port is split as equally as possible between the two subrequest sets.

flow id	IP	input port	output port	OP	bandwidth	favors	bandwidth in B1	bandwidth in B2
f1		in1	out1		9	B1	5	4
f2		in1	out2		1	B2	0	1
f3		in1	out3		11	B1	6	5
f4		in1	out4		6		3	3
f5		in2	out1		5	B2	2	3
f6		in2	out2		15	B1	8	7
f7		in2	out3		11	B1	6	5
f8		in3	out1		13	B1	7	6
f9		in3	out1		5	B2	2	3
f10		in3	out3		9	B2	4	5
f11		in3	out4		2		1	1
f12		in3	out4		3	B1	2	1
f13		in4	out4		15	B2	7	8
f14		in4	out4		3	B1	2	1

key

even-bandwidth request
odd-bandwidth request that favors B1
odd-bandwidth request that favors B2

pairs

unpaired

Table 5: Calculation of bandwidth subrequest sets *B1* and *B2*.

port	load in B	load in B1	load in B2
in1	27	14	13
in2	31	16	15
in3	32	16	16
in4	18	9	9
out1	32	16	16
out2	16	8	8
out3	31	16	15
out4	29	15	14

Table 6: Port loads in *B1* and *B2*.

Recursive splitting would divide *B1* into two subframes, each having load 14/2 = 7 for in1 and load 16/2 = 8 for out1, and so on. Similarly, *B2* would be divided into two subframes, where each subframe would have load 16/2 = 8 for out1 and where one subframe would have load (13+1)/2 = 7 for in1 and the other subframe would have load (13-1)/2 = 6 for in1. Five levels of recursive splitting will split *B* into 32 request sets such that no port has load greater than 1 in any single request set. At that point the switch can satisfy the requests during a single time slot.

This completes the description of the basic recursively balanced scheduling method.

4. Degree of smoothness achieved by the basic method

Smooth schedules were described above as ones in which the scheduled time slots for each flow are approximately uniformly distributed throughout the duration of the scheduling frame. Here is a quantitative characterization of the sense in which recursively balanced schedules are smooth.

Suppose we have a schedule *S* for a frame of size *n*. For some flow or port whose load is of interest, the question is: How is the load spread out over the schedule? Let the load be *m* cells per frame. Some slots in *S* contain a unit of the load and the other slots contain no load. From the point of view of the flow or port, the slots in *S* that contain a unit of load are *full* and the rest are *empty*.

Now let *S** be the schedule produced by concatenating an unlimited number of repetitions of *S* and consider any interval I consisting of $|I|$ consecutive slots from *S**. Define the perfect load in *I*, $P(I)$, as $|I|m/n$ and the scheduled load in *I*, $S(I)$, as the number of full slots in *I*. If the schedule were perfectly smooth, then we would expect $P(I)=S(I)$. This cannot usually be achieved, if only because slots are discrete and $|I|m/n$ may not be an integer. We define the *scheduling discrepancy* (with regard to a flow or port) over interval I as $|S(I)-P(I)|$. We define the *maximum scheduling discrepancy* in *S*, $msd(S)$, as the maximum scheduling discrepancy over any interval *I* from *S**.

If no two flows use the same input or the same output, it is possible to minimize the maximum scheduling discrepancies for all flows simultaneously, producing a schedule (an *all-min-msd schedule*) in which the maximum scheduling discrepancy for each flow is less than 1 cell. However, given a bandwidth request set that includes flows with input or output conflicts, an all-min-msd schedule is not necessarily possible. Consider, for example, scheduling the bandwidth request set shown in Table 7 into a frame of four slots. With this frame size, a flow with bandwidth 2 has a minimum maximum scheduling discrepancy of ½, and to achieve this, the flow must be scheduled into every other slot. Without loss of generality, let us assume that f1 is scheduled into slots 1 and 3, as illustrated in Figure 4. Then f2 must be scheduled into slots 2 and 4. Since f3 shares out1 with f1, it follows that f3 must be scheduled into an even-numbered slot. Similarly, f4 must be scheduled into an odd-numbered slot. Both f3 and f4 share in4 with f5. This leaves f5 with one even-numbered and one odd-numbered slot, resulting in a maximum scheduling discrepancy of 1 for f5.

flow id	input port	output port	bandwidth
f1	in1	out1	2
f2	in1	out2	2
f3	in4	out1	1
f4	in4	out2	1
f5	in4	out4	2

Table 7: A bandwidth request set with no all-min-msd schedule in a frame of four slots.

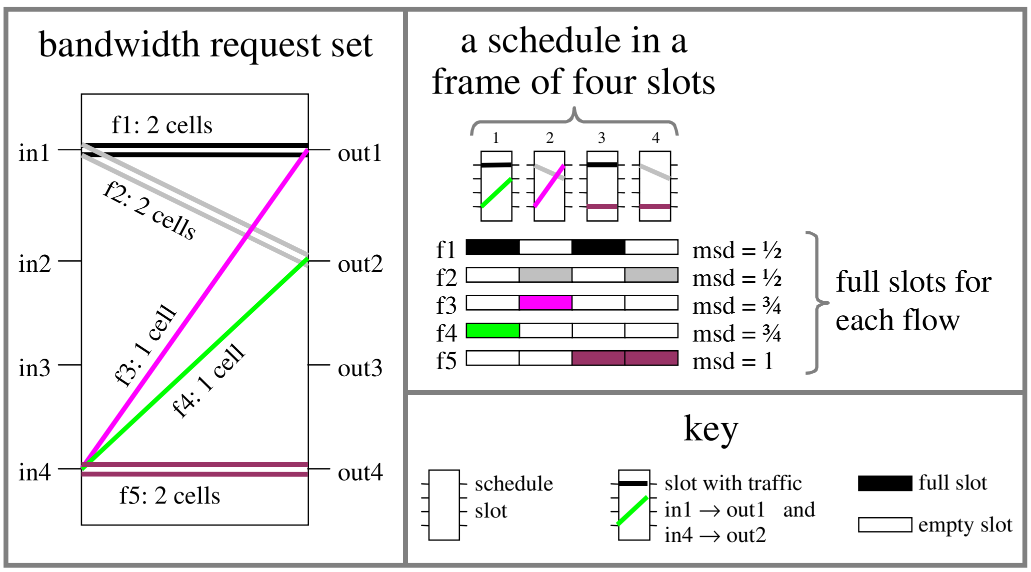

Figure 4: A four-slot schedule for the bandwidth request set of Table 7.

Since an operational system recomputes the schedule from time to time as flows are added or removed, we extend the definition of scheduling discrepancy to a concatenation in which the schedules need not all be identical. Given a set SS of schedules, we define the maximum scheduling discrepancy in SS, $msd(SS)$, as the maximum scheduling discrepancy over any interval I from any concatenation of schedules from SS. We are particularly interested in sets of recursively balanced schedules, although for comparison we also consider sets of all legal schedules.

Let LS_n be the set of all legal schedules with a frame of size n. Let RB_n be the set of all recursively balanced schedules with frame size n and let $RB_{n,m}$ be the subset of those schedules that have load m for the flow or port of interest. A recursively balanced schedule has the following smoothness property with regard to the load of that flow or port: a load of m cells per frame has been split in half over and over again as equally as possible. Using this property we will obtain bounds on $msd(RB_n)$ and $msd(RB_{n,m})$.

In Section 4.1 we give a method and example of calculating the maximum scheduling discrepancy in any given schedule S. In Section 4.2 we compute the value of $msd(LS_n)$ by considering pathological but legal schedules. Generalizing the method of Section 4.1 and applying the smoothness property of recursively balanced schedules, in Section 4.3 we obtain a bound on $msd(RB_n)$ when n is a power of two. In Section 4.4 we compare this bound with some actual values of $msd(RB_{n,m})$.

4.1. Maximum scheduling discrepancy in a given schedule

Suppose we have a scheduling frame of size n and a flow or port with a load m. The maximum scheduling discrepancy in any given schedule S for this situation can be calculated as follows. First we construct two functions from cumulative (elapsed) slots to cumulative cells. The cumulative scheduled load function $\overline{S}(t)$ is a bumpy line with a slope of 0 during each empty slot and a slope of 1 during each full slot. The cumulative perfect load function $\overline{P}(t)$ is a straight line with a slope of m/n. Both $\overline{S}$ and $\overline{P}$ start at $(0,0)$ and end at (n,m): $\overline{S}(0)=\overline{P}(0)=0$ and $\overline{S}(n)=\overline{P}(n)=m$. Next we compute the difference $\overline{D}(t)\equiv\overline{S}(t)-\overline{P}(t)$. The difference between the extreme high and low values of $\overline{D}$ is the maximum scheduling discrepancy in the schedule S. The interval that exactly spans the times at which the extreme high and low values of $\overline{D}$ occur has this discrepancy, and no interval has a greater discrepancy. Note that this calculation applies to any schedule, regardless of whether or not it is recursively balanced.

An example will illustrate the calculation. Figure 5 shows a construction of a recursively balanced schedule for frame size $n=16$ and load $m=10$. Note that each time the load splits, the division is as equal as possible; hence the schedule is recursively balanced. Full slots are shaded in and empty slots are shown empty.

Figure 6 shows the cumulative scheduled load function $\overline{S}$ and cumulative perfect load function $\overline{P}$ for this schedule.

The difference $\overline{D}\equiv\overline{S}-\overline{P}$ is plotted in Figure 7. An interval I of consecutive slots can be represented by its beginning and ending cumulative slot values, say t_0 and t_1. Note that $t_1-t_0=|I|$. Observe that $\overline{S}(t_1)-\overline{S}(t_0)$ is exactly the number of full slots in I and $\overline{P}(t_1)-\overline{P}(t_0)=|I|m/n$. Hence the scheduling discrepancy over I is $|\overline{D}(t_1)-\overline{D}(t_0)|$.

From this illustration it is easy to see that the maximum scheduling discrepancy over any interval within any concatenation of repetitions of S is indeed the difference between the extreme high and low values of $\overline{D}$. For the example in Figure 7, the maximum scheduling discrepancy is 2.25 cells.

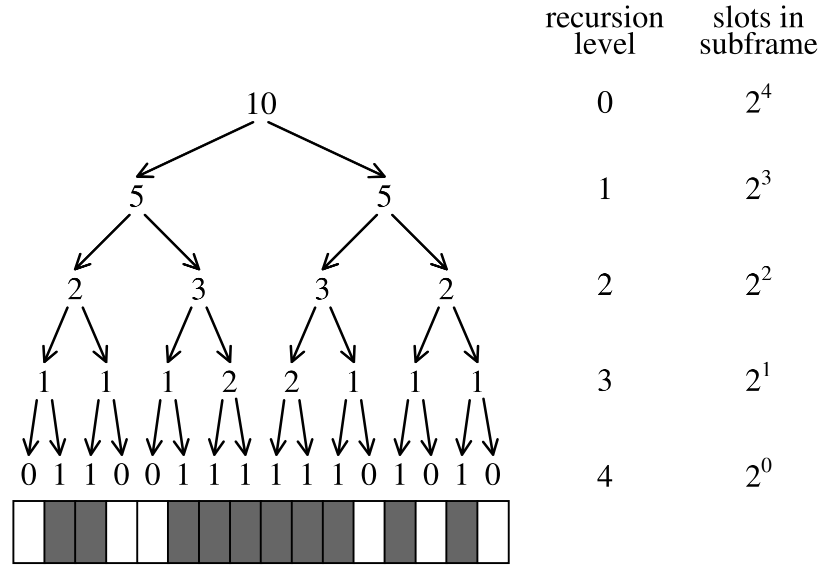

Figure 5: A recursively balanced schedule for load 10 in frame size 16.

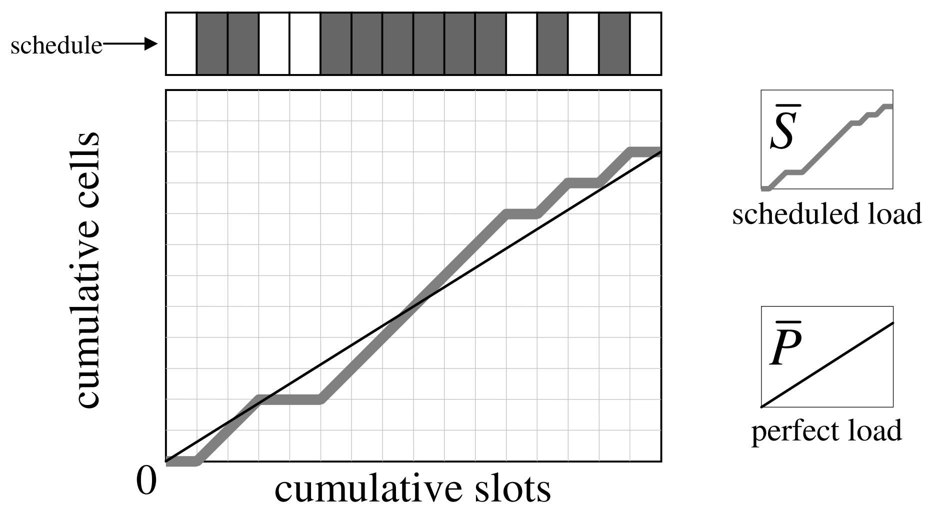

Figure 6: Scheduled load and perfect load for one frame of a recursively balanced schedule.

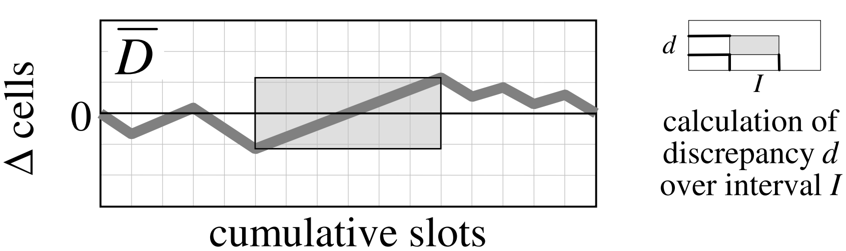

Figure 7: Difference between scheduled load and perfect load.

4.2. Maximum scheduling discrepancy in legal schedules

In this section we compute $msd(LS_n)$, the maximum scheduling discrepancy in any concatenation of legal schedules of any loads in frames of size *n*. First, let us consider legal schedules of load *m*. Since $\overline{D} \equiv \overline{S} - \overline{P}$, the highest possible value of $\overline{D}$ results from packing the entire load into the initial slots of the scheduling frame and evaluating at time *m*. Using the subscript *m* to indicate this schedule, we have

$$\overline{S}_m(m) = m\,,$$

$$\overline{P}_m(m) = m \cdot m/n\,, \text{ and}$$

$$\overline{D}_m(m) = (m \cdot n - m \cdot m)/n\,.$$

Next, keeping the frame size fixed at *n*, $\overline{D}_m(m)$ is maximized when $m = n/2$. Hence, for any time *t* in any legal schedule for any load in a frame of size *n*,

$$\overline{D}(t) \leq n/4\,,$$

and this bound is tight. A symmetrical argument packing the entire load into the final slots of the scheduling frame bounds $\overline{D}$ from below. Hence, the maximum scheduling discrepancy in any concatenation of legal schedules with a frame of size *n* is $n/2$. Formally,

$$msd(LS_n) = n/2\,.$$

For example, suppose we have a 1024-slot scheduling frame (*n* = 1024). Then the maximum scheduling discrepancy in any concatenation of legal schedules is 512 cells. This maximum is attained in the case of a load of 512 cells per frame and two consecutive scheduling frames. In the first frame all 512 cells of the load appear in the final 512 slots of the frame, leaving the initial 512 slots empty. In the second frame (presumably due to an unfortunate scheduling change) all 512 cells of the load appear in the initial slots of the frame, leaving the final 512 slots empty. The 1024-slot interval consisting of the final 512 slots of the first frame and the initial 512 slots of the second frame has a scheduled load of 1024 and a perfect load of 512, for a discrepancy of 512 cells.

Although the recursively balanced scheduling method does not necessarily minimize the maximum scheduling discrepancies of all flows and ports, the discrepancies it produces are much smaller than the maximum scheduling discrepancies of schedules that are legal but satisfy no additional smoothness requirements. Consider again a 1024-slot scheduling frame. In the following section, we will show that the maximum scheduling discrepancy in any concatenation of 1024-slot recursively balanced schedules of any loads cannot exceed 6.89 cells. This is much better than the 512 cell maximum discrepancy in legal schedules.

4.3. A bound for msd in recursively balanced schedules

In this section we demonstrate that maximum scheduling discrepancy in recursively balanced schedules is bounded by the inequality

$$msd\left(RB_{2^k}\right) \le 2\left(\frac{3k+1-\left(-½\right)^k}{9}\right) = \frac{2k}{3}+2\left(\frac{1-\left(-½\right)^k}{9}\right).$$

That is, in any concatenation of recursively balanced 2^k-slot scheduling frames of any loads, the scheduling discrepancy over any interval is at most $2k/3+\varepsilon_k$ cells, where $0 \le \varepsilon_k \le 1/3$.

Consider a concatenation of recursively balanced 2^k-slot scheduling frames. Let $n = 2^k$ be the number of slots in a frame. Recall the definitions of $\overline{S}(t)$, $\overline{P}(t)$, and $\overline{D}(t)$ from Section 4.1. Note that $\overline{P}(t)$ has constant slope during a frame. Our analysis depends on finding bounds for $\overline{D}(t)$.

Since each whole scheduling frame has the correct number of slots allocated for its load, it follows that at each frame boundary—that is, whenever *t* is an integer multiple of the frame size *n*—we have $\overline{S}(t)=\overline{P}(t)$, and thus $\overline{D}(t)=0$.

The recursively balanced scheduling method proceeds by splitting the frame into smaller and smaller subframes. Let us consider any frame or subframe *F* starting at time t_0 and ending at time at t_1, where $t_1-t_0 \ge 2$. *F* contains t_1-t_0 slots. The defining smoothness property of a recursively balanced schedule is that the load in each half of *F* differs from half of the load of *F* by at most half a cell. In particular, the load in the first half of *F* can be no greater than half the load of *F* plus half a cell:

$$\overline{S}\left(\frac{t_1-t_0}{2}\right)-\overline{S}(t_0) \le \frac{\overline{S}(t_1)-\overline{S}(t_0)+1}{2},$$

or equivalently

$$\overline{S}\left(\frac{t_1-t_0}{2}\right) \le \frac{\overline{S}(t_1)+\overline{S}(t_0)+1}{2}.$$

Within a frame, $\overline{P}(t)$ has constant slope. So we have

$$\overline{P}\left(\frac{t_1-t_0}{2}\right) = \frac{\overline{P}(t_1)+\overline{P}(t_0)}{2}.$$

Hence we can bound the value of $\overline{D}$ at the midpoint of *F*:

$$\begin{aligned}
\overline{D}\left(\frac{t_1-t_0}{2}\right) &= \overline{S}\left(\frac{t_1-t_0}{2}\right)-\overline{P}\left(\frac{t_1-t_0}{2}\right) \\
&\le \frac{\overline{S}(t_1)+\overline{S}(t_0)+1}{2} - \frac{\overline{P}(t_1)+\overline{P}(t_0)}{2} \\
&= \frac{\overline{S}(t_1)-\overline{P}(t_1)+\overline{S}(t_0)-\overline{P}(t_0)+1}{2} \\
&= \frac{\overline{D}(t_1)+\overline{D}(t_0)+1}{2}.
\end{aligned}$$

Let i and j be integers, where $0 \le i \le k$. Observe that any time of the form $j2^{k-i}$ is the boundary of a subframe of size 2^{k-i}. Let us define $\hat{D}_i$ as the maximum value of $\overline{D}$ at any boundary of a subframe of size 2^{k-i}. Formally,

$$\hat{D}_i \equiv \max_j \left\{ \overline{D}\left(j2^{k-i}\right) \right\}.$$

Now any time of the form $j2^k$ is a frame boundary, hence

$$\hat{D}_0 \;=\; 0.$$

Any time of the form $j2^{k-1}$ is either a frame boundary (for which $\overline{D}$ has value 0) or the midpoint of a frame (for which, from the midpoint inequality above, $\overline{D}$ has value at most $(0+0+1)/2$). Hence

$$\hat{D}_1 \;\le\; 1/2.$$

For $2 \le i \le k$, any time of the form $j2^{k-i}$ can be characterized based on whether j is even or odd. If j is even, $j2^{k-i}$ is an integer multiple of $2^{k-(i-1)}$. If j is odd, $j2^{k-i}$ is the midpoint of a subframe whose starting and ending times are successive integer multiples of $2^{k-(i-1)}$, one of which is necessarily an integer multiple of $2^{k-(i-2)}$. Hence, for $2 \le i \le k$,

$$\hat{D}_i \;\le\; \max\left(\hat{D}_{i-1}, \quad \frac{\hat{D}_{i-1} + \hat{D}_{i-2} + 1}{2} \right).$$

From the above recurrence, it follows (as can readily be checked) that

$$\hat{D}_i \;\le\; \frac{3i + 1 - \left(-½\right)^i}{9},$$

for any integer i in the range $0 \le i \le k$. Taking $i = k$, we have

$$\overline{D}(t) \;\le\; \hat{D}_k \;\le\; \frac{3k + 1 - \left(-½\right)^k}{9},$$

for any integer t, and thus for any time t whatsoever. (Since both $\overline{S}$ and $\overline{P}$ have constant slope within any single slot, the maximum value of $\overline{D}$ must occur at a slot boundary.)

By a symmetrical argument, $\overline{D}$ is bounded below by

$$\overline{D}(t) \;\ge\; -\frac{3k + 1 - \left(-½\right)^k}{9},$$

and the bound on maximum scheduling discrepancy claimed at the start of this section,

$$msd\left(RB_{2^k}\right) \;\le\; 2\left(\frac{3k + 1 - \left(-½\right)^k}{9} \right),$$

follows immediately.

Table 8 shows the numerical value of this bound for various values of k. Observe from the table that the maximum scheduling discrepancy cannot exceed 2.88 cells in any set of recursively balanced schedules whose frame size is 16 slots. In Section 4.1 we calculated a maximum scheduling discrepancy of 2.25 cells in a particular recursively balanced schedule of this frame size. As expected, 2.25 does not exceed the theoretical bound of 2.88.

As discussed later in Section 7, our switch implementation uses a frame size of 1024 slots. From Table 8 we see that the maximum scheduling discrepancy in recursively balanced schedules of this frame size cannot exceed 6.89 cells.

k	frame size $n = 2^k$	bound on $msd(RB_n)$, $2\left(\frac{3k+1-(-½)^k}{9}\right)$, rounded up to nearest 0.01
0	1	0.00
1	2	1.00
2	4	1.50
3	8	2.25
4	16	2.88
5	32	3.57
6	64	4.22
7	128	4.90
8	256	5.56
9	512	6.23
10	1024	6.89

Table 8: Bound on $msd(RB_n)$ for various frame sizes.

4.4. Actual msd values for certain loads

Recall that we have a scheduling frame of size $n = 2^k$ and a flow or port of interest whose load is m. In the previous section we obtained a bound for $msd(RB_{2^k})$, the maximum scheduling discrepancy in any concatenation of recursively balanced 2^k-slot scheduling frames of any loads. For a given load m, the maximum scheduling discrepancy $msd(RB_{2^k,m})$ could be much less than the bound.

In general, the worst case discrepancy for any given load occurs when a frame with its load scheduled as late as permissible (called a *latest-load* frame) is followed—after zero or more intervening frames—by a frame with its load scheduled as early as permissible (an *earliest-load* frame). A latest-load frame achieves the minimum possible value of $\overline{D}(t)$ and an earliest-load frame the maximum possible value. Some other schedules may also achieve these extremal values, but none can exceed them.

Let us consider some examples.

When $m = 1$, we have

$$msd\left(RB_{2^k,1}\right) = 2\left(1 - \frac{1}{2^k}\right).$$

The latest-load frame has its last slot full, and the earliest load frame has its first slot full.

When m is a power of 2, say $m = 2^a$, we have

$$msd\left(RB_{2^k,2^a}\right) = 2\left(1 - \frac{1}{2^{k-a}}\right) < 2.$$

Any recursively balanced schedule for this load puts one cell of load into each 2^{k-a}-slot subframe. This is exactly the same situation as a load of 1 in a frame of size 2^{k-a}.

Now suppose m is a sum of two distinct powers of 2, say $m = 2^a + 2^b$, with $a < b$. Observe that the full slots in any recursively balanced schedule can be partitioned into two sets *A* and *B*, where *A* is the set of full slots in some recursively balanced schedule of load 2^a and *B* is the set of full slots in some recursively balanced schedule of load 2^b. Hence,

$$msd\left(RB_{2^k,2^a+2^b}\right) \le msd\left(RB_{2^k,2^a}\right) + msd\left(RB_{2^k,2^b}\right).$$

$$= 2\left(1 - \frac{2^a}{2^k} + 1 - \frac{2^b}{2^k}\right) < 4.$$

This bound is not tight because, for example, *A* and *B* cannot both contain the first slot of any subframe. In fact, the worst case discrepancy is

$$msd\left(RB_{2^k,2^a+2^b}\right) = 2\left(1 - \frac{2^a}{2^k} + 1 - \frac{2^b}{2^k} - \frac{2^a}{2^b}\right).$$

The latest-load frame has two full slots in its last 2^{k-b}-slot subframe and a full slot as the last slot of its next-to-last 2^{k-b}-slot subframe, making a total of three cells of load in the last $2^{k-b}+1$ slots of the frame. A symmetrical situation occurs at the beginning of the earliest-load frame.

Although it is beyond the scope of this report to give a complete analysis, maximum scheduling discrepancies are small for loads which are sums and differences of a small number of powers of two.

As discussed later in Section 7, our switch implementation uses a frame size of 1024 slots. Table 9 gives maximum scheduling discrepancies for arbitrary concatenations of recursively balanced schedules of any given load $m = a + b$ with frame size 1024. We show the *msd* values only for loads up to a few more than 512. Values for higher loads can be found by symmetry: because the full slots in an earliest-load frame of load *m* are precisely the empty slots in a latest-load frame of load $n - m$, and vice versa, $msd\left(RB_{n,m}\right) = msd\left(RB_{n,n-m}\right)$.

In a frame of size 1024, the worst-case maximum scheduling discrepancy of 6.89 occurs for a load of $341 = 101010101_2$ cells per frame (and also for $1024 - 341$ cells per frame). This value, highlighted in Table 9, is exactly the worst-case bound derived in Section 4.3.

a \ b		0 0000_2	1 0001_2	2 0010_2	3 0011_2	4 0100_2	5 0101_2	6 0110_2	7 0111_2	8 0000_2	9 1001_2	10 1010_2	11 1011_2	12 1100_2	13 1101_2	14 1110_2	15 1111_2
0	0_2	0.00	2.00	2.00	2.99	1.99	3.49	2.99	3.49	1.98	3.73	3.48	4.23	2.98	4.22	3.47	3.72
16	10000_2	1.97	3.84	3.71	4.59	3.46	4.83	4.21	4.58	2.95	4.58	4.20	4.82	3.45	4.57	3.69	3.81
32	100000_2	1.94	3.87	3.81	4.74	3.68	5.12	4.55	4.99	3.42	5.11	4.79	5.48	4.16	5.35	4.54	4.72
48	110000_2	2.91	4.72	4.53	5.34	4.15	5.46	4.77	5.08	3.39	4.95	4.51	5.07	3.63	4.69	3.75	3.81
64	1000000_2	1.88	3.84	3.81	4.78	3.74	5.21	4.68	5.14	3.61	5.33	5.04	5.76	4.48	5.69	4.91	5.13
80	1010000_2	3.34	5.19	5.03	5.87	4.71	6.05	5.39	5.74	4.08	5.67	5.26	5.85	4.45	5.54	4.63	4.72
96	1100000_2	2.81	4.72	4.62	5.53	4.43	5.83	5.24	5.64	4.05	5.70	5.36	6.01	4.66	5.82	4.97	5.13
112	1110000_2	3.28	5.06	4.84	5.62	4.40	5.68	4.96	5.24	3.52	5.04	4.57	5.10	3.63	4.66	3.69	3.72
128	10000000_2	1.75	3.73	3.71	4.70	3.68	5.16	4.64	5.13	3.61	5.34	5.07	5.81	4.54	5.77	5.00	5.24
144	10010000_2	3.47	5.33	5.18	6.04	4.90	6.26	5.61	5.97	4.33	5.94	5.54	6.15	4.76	5.87	4.97	5.08
160	10100000_2	3.19	5.11	5.03	5.95	4.87	6.29	5.71	6.13	4.55	6.22	5.89	6.56	5.23	6.40	5.57	5.74
176	10110000_2	3.91	5.70	5.50	6.29	5.09	6.38	5.68	5.97	4.27	5.81	5.36	5.90	4.45	5.49	4.54	4.58
192	11000000_2	2.63	4.58	4.53	5.48	4.43	5.88	5.33	5.78	4.23	5.94	5.64	6.34	5.04	6.24	5.44	5.64
208	11010000_2	3.84	5.67	5.50	6.32	5.15	6.47	5.80	6.13	4.45	6.03	5.61	6.18	4.76	5.83	4.91	4.99
224	11100000_2	3.06	4.95	4.84	5.73	4.62	6.01	5.39	5.78	4.17	5.81	5.45	6.09	4.73	5.87	5.00	5.14
240	11110000_2	3.28	5.04	4.81	5.57	4.34	5.60	4.86	5.13	3.39	4.90	4.42	4.93	3.45	4.46	3.47	3.49
256	100000000_2	1.50	3.49	3.48	4.47	3.46	4.95	4.44	4.93	3.42	5.16	4.90	5.64	4.38	5.62	4.86	5.10
272	100010000_2	3.34	5.21	5.07	5.94	4.80	6.17	5.54	5.90	4.27	5.88	5.50	6.11	4.73	5.84	4.96	5.07
288	100100000_2	3.19	5.12	5.04	5.97	4.90	6.33	5.75	6.18	4.61	6.29	5.96	6.64	5.32	6.50	5.68	5.85
304	100110000_2	4.03	5.83	5.64	6.44	5.24	6.54	5.85	6.15	4.45	6.01	5.56	6.11	4.66	5.72	4.77	4.82
320	101000000_2	2.88	4.83	4.79	5.75	4.71	6.17	5.63	6.09	4.55	6.26	5.96	6.67	5.38	6.59	5.80	6.01
336	101010000_2	4.22	6.05	5.89	6.72	5.55	$\boxed{6.89}$	6.22	6.56	4.89	6.47	6.06	6.64	5.23	6.31	5.39	5.48
352	101100000_2	3.56	5.46	5.36	6.25	5.15	6.54	5.94	6.34	4.73	6.38	6.03	6.67	5.32	6.47	5.61	5.76
368	101110000_2	3.91	5.68	5.45	6.22	4.99	6.26	5.54	5.81	4.08	5.60	5.12	5.64	4.16	5.19	4.21	4.23
384	110000000_2	2.25	4.22	4.20	5.17	4.15	5.62	5.10	5.57	4.05	5.77	5.50	6.22	4.95	6.17	5.39	5.62
400	110010000_2	3.84	5.69	5.54	6.39	5.24	6.59	5.94	6.29	4.64	6.24	5.84	6.44	5.04	6.14	5.24	5.34
416	110100000_2	3.44	5.35	5.26	6.17	5.09	6.50	5.91	6.32	4.73	6.40	6.06	6.72	5.38	6.54	5.71	5.87
432	110110000_2	4.03	5.82	5.61	6.39	5.18	6.47	5.75	6.04	4.33	5.87	5.40	5.94	4.48	5.51	4.55	4.59
448	111000000_2	2.63	4.57	4.51	5.46	4.40	5.84	5.29	5.73	4.17	5.87	5.56	6.25	4.95	6.14	5.33	5.53
464	111010000_2	3.72	5.54	5.36	6.17	4.99	6.31	5.63	5.95	4.27	5.83	5.40	5.97	4.54	5.61	4.68	4.74
480	111100000_2	2.81	4.69	4.57	5.46	4.34	5.72	5.10	5.48	3.86	5.49	5.12	5.75	4.38	5.51	4.64	4.78
496	111110000_2	2.91	4.66	4.42	5.17	3.93	5.19	4.44	4.70	2.95	4.46	3.96	4.47	2.98	3.98	2.99	2.99
512	1000000000_2	1.00	2.99	2.99	3.98	2.98	4.47	3.96	4.46	2.95	4.70	4.44	5.19	3.93	5.17	4.42	4.66

Table 9: Values of $msd(RB_{1024,a+b})$, rounded to the nearest 0.01.

Because $msd(RB_{2n,2m}) = msd(RB_{n,m})$, Table 9 can also be used to look up the maximum discrepancy values for any given load in a 2^k-slot frame, where $2^k < 1024$. For example, for a load of 21 cells and a frame size of 64 slots, we have

$$msd(RB_{21,64}) = msd(RB_{16\times 21,16\times 64}) = msd(RB_{336,1024}) \approx 4.22\,.$$

Note that this value, being the largest in the first column of the table, is the worst-case maximum scheduling discrepancy for any load in a 64-slot frame.

5. Output link smoothness

In Section 1, we mentioned that a single output port of a switch may be used to drive several *communication links* that each run at a lower bandwidth than the per-port bandwidth of the switch. In such cases it is desirable that the slots containing flows destined for any given output link be spread out smoothly over the scheduling frame, in order to decrease both the amount of output buffering memory required and the amount of time data spends in output buffers.

The basic recursively balanced scheduling method can be extended to achieve per-output-link smoothing. We modify each bandwidth request b by replacing the output port field $b.o$ with an output link field $b.ol$ that gives the specific output link. All references to output port $b.o$ are interpreted by mapping output link $b.ol$ to the output port that contains it. We define $load(B, ol)$, the *load* in B on output link ol, as the sum of the bandwidths for all bandwidth requests with output link ol:

$$load(B, ol) \equiv \sum_{b \in B: b.ol = ol} b.m\ .$$

Note that a feasible schedule is not possible if any output link is overloaded. To the properties SPLIT.1-5 from Section 3.1 we add the following properties:

SPLIT.6. Each flow f has the same output link in $B1$ and $B2$ as in B. Formally,

if $B1[f]$ exists then $B1[f].ol = B[f].ol$, and

if $B2[f]$ exists then $B2[f].ol = B[f].ol$.

SPLIT.7. Each output link ol has its load divided as equally as possible. Formally,

$$\left|load(B1, ol) - load(B2, ol)\right| \leq 1\ .$$

Property SPLIT.6 amplifies SPLIT.1 to guarantee that the splitting step preserves each flow's output link. Property SPLIT.7 guarantees per-output-link smoothness. Note that SPLIT.7 does not supersede property SPLIT.5. Dividing the load on each output link as equally as possible does not by itself guarantee that the load on each output port will be divided as equally as possible, which is what SPLIT.5 requires. The problem is that each of the output links of the same output port could favor the same subrequest set, leading to an excessive imbalance in the division of the load on the output port.

The enhanced properties SPLIT.1-7 can be satisfied by refining the way in which the output pairing OP is computed. Define the *output link pairing*, OLP, as the subset of OP in which the members of a pair have the same output link:

$$OLP \equiv \left\{\{b, c\} \in OP : b.ol = c.ol\right\}.$$

In addition to properties OUTPAIR.1-3 we require:

OUTPAIR.4. For any output link ol, there is at most one odd-bandwidth request for ol that does not appear in OLP.

Such a pairing may be found by first pairing requests that share output links until each output link has at most one unpaired request, then pairing as-yet-unpaired requests that share output ports until each output port has at most one unpaired request.

In Section 4 we derived a bound on the maximum scheduling discrepancy of any recursively balanced schedule produced by the basic method. The same analysis applies to the extended method, yielding a bound on the maximum scheduling discrepancy for an output link. This discrepancy is an upper bound on the amount of memory in an output link's buffer needed to accommodate the lack of perfect smoothness in the schedule. In the worst case, for a recursively balanced schedule with a frame size n and load m, the link removes cells from the output buffer at exactly the minimum necessary rate of m/n cells per slot, for a total of $|I|m/n$ cells over any interval I. Over the same interval, as many cells as the number of full cells in I may arrive at the output buffer. Hence $msd(RB_{n,m})$ bounds the difference that must be absorbed in the output buffer.

6. Odd-size frames

The recursively balanced scheduling method described above uses a splitting step that splits the bandwidth request set for an even-size frame into subrequest sets for two equal sized subframes. Since the splitting step is applied recursively, the size of each subframe to be split must also be even. Hence, as given, the method applies only to frames whose size (in slots) is an integral power of two. At the cost of losing some smoothness in the resulting schedules, the method can be extended to schedule frames of arbitrary size by generalizing the splitting step to apply to odd-size frames.

Being able to handle arbitrary-size frames is useful for several reasons. It might be desirable to use a non-power-of-two frame in order to support certain exact bandwidth requirements. Even in the case of a power-of-two frame, it might be desirable to reserve some slots throughout the schedule for other purposes, thus in effect leaving a non-power-of-two frame for the recursively balanced scheduling method. For example, slots might be reserved in order to handle multicast traffic, as in our implementation described in Section 7.

Another reason for reserving slots would be to improve the performance of opportunistic traffic by leaving some slots completely free of scheduled traffic. This is a fairly interesting effect. The smooth schedule (by design) spreads its port usage throughout the slots it uses. This tends to increase the probability that a cell of opportunistic traffic will find either its input port or its output port in use in any given slot. By reserving some slots, the scheduled traffic is compressed into fewer slots, thus making it more likely that opportunistic cells can be forwarded.

Figure 8 shows an example of this effect in a frame of 8 slots. The bandwidth request set uses in1 four times and out1 three times. Case (a) shows a smooth schedule in which these uses are spread out over seven slots, leaving only one slot available for opportunistic traffic from in1 to out1. Although this schedule is pathological from this point of view, it is certainly a perfectly fine smooth schedule and it could have been produced by the recursively balanced scheduling algorithm. In case (b), four slots have been reserved for opportunistic traffic. The bandwidth request set is feasible for the remaining frame of four slots, so a smooth schedule can be constructed. In this case the four reserved slots are available for opportunistic traffic from in1 to out1.

To split a bandwidth request set B into a frame F of odd size $2k+1$, we proceed as follows:

1. Split the frame F into three subframes, *F1*, *F2*, *F3*, where *F1* and *F3* have size k and *F2* consists of a single slot.
2. Allocate one slot's worth of sufficiently many non-conflicting flows to the single-slot subframe *F2* so that the remaining requests are feasible for $2k$ slots.
3. Split the remaining requests into subrequest sets for *F1* and *F3* using the even-size frame splitting method discussed in Section 3.2.

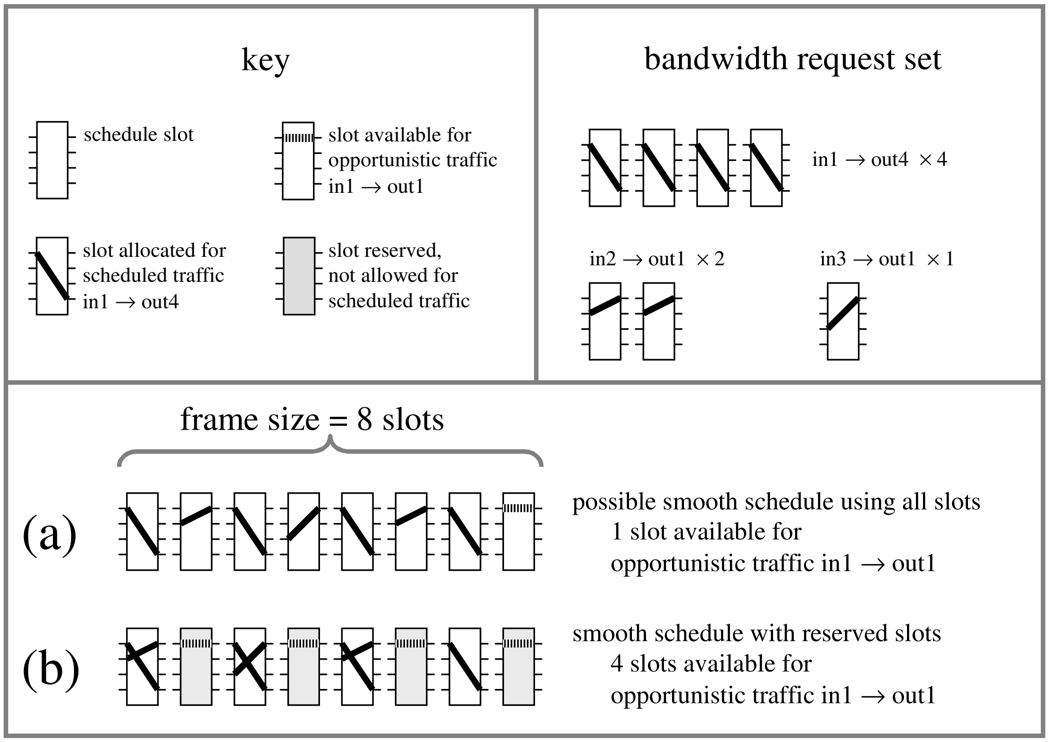

Figure 8: Reserving slots can improve the performance of opportunistic traffic.

The flows assigned to $F2$ in step 2 may be simply the flows assigned to any single slot of any legal (not necessarily smooth) schedule for the bandwidth request set B in $2k+1$ slots. One way of obtaining such a set of flows is to compute a *maximum matching* [4, pp. 600-604] in the bipartite graph whose vertices are the input ports and output ports of the switch and whose edges are those input-output pairs for which there is at least one positive bandwidth flow. Our implementation uses a different method, described in Section 7.5.

7. Implementation

We have implemented the recursively balanced scheduling method as part of the control software on the AN2 switch [1, 14, 16]. An AN2 network has been in daily use as a service network at our laboratory, the Digital Equipment Corporation Systems Research Center, since the end of 1994.

7.1. Hardware description

The AN2 switch is a sixteen-port, input-buffered, ATM crossbar switch designed and built at our laboratory. Each of the ports has a bandwidth of 800 Megabits/second, for a total aggregate switch bandwidth of 12.8 Gigabits/second. Since the bit rate numbers in the ATM arena often cover different amounts of overhead, matters can be clarified by speaking in terms of ATM cells per second. The bandwidth of each AN2 switch port is 1923 kilocells/second.

The AN2 switch hardware provides two service classes: *scheduled* and *opportunistic*. Scheduled traffic has absolute priority and is sent across the crossbar according to a schedule provided by software. The AN2 schedule frame contains 1024 cell slots. Each slot contains an entry for each of the sixteen input ports listing which virtual circuit has priority in that schedule slot from that input port. The scheduling software must guarantee that there is no collision of output ports. If the listed virtual circuit has no cells available, its input and output ports remain available for opportunistic traffic. Schedule slots for an input port are left unused by listing virtual circuit zero, which never has cells. Opportunistic traffic is sent across the crossbar on a best-effort basis with hardware matching of available input and output ports [15]. To avoid head-of-line blocking [11], each input port maintains separate cell queues for each output port.

Each AN2 switch port accommodates an input/output line card that interfaces the switch port to one or more external communication links. The *quad line card* supports four SONET OC-3c links, multiplexing input cells to the switch port and demultiplexing output cells. A SONET OC-3c link (so-called 155 Megabits/second) carries 353 Kilocells/second. A line card that supports a single SONET OC-12 link has been developed [5], but none of these are installed at our laboratory.

Each link on the quad line card contains a small FIFO output buffer for the purpose of matching the high cell-rate output of the switch port to the slower link rate. Ten cells of storage in each FIFO are dedicated to scheduled traffic. Whenever a speed-matching FIFO gets too full, the switch hardware automatically blocks opportunistic traffic destined to that output link (causing it to queue up in the switch's input buffers). In order to provide guaranteed service, the hardware never blocks scheduled traffic. Hence the schedule must be approximately smooth with respect to each output link or else the scheduled traffic could overrun its dedicated storage.

A scheduled virtual circuit can forward cells simultaneously to multiple output ports and links. This is how the switch hardware supports multicast. The hardware does not support multicast for opportunistic traffic.

7.2. Software environment

The switch control software maintains a list of all scheduled virtual circuits, listing for each such circuit the input port, the output link, and the required bandwidth in cells per frame. (The output port number can be determined algebraically from the output link number.) A special output link value is used to indicate a multicast circuit. A multicast circuit that sends to all output links is *full broadcast*; if it sends to a proper subset there may be output ports that it does not use. Although one could exploit a multicast circuit's unused output ports, our current system software does not do so. We treat all multicast circuits as if they were full broadcast when constructing the schedule. This also allows multicast circuits to add and drop output links without requiring the schedule to be modified. We do not expect that multicast traffic will find heavy use in the network. Some heuristics are known for scheduling multicast circuits under the TDMA model [3].

Requests to add and delete scheduled virtual circuits are checked for admissibility and then batched for processing. The admission test checks the input and output loads to make sure that the schedule would be feasible and also verifies that the total number of scheduled circuits would not exceed a software implementation limit. The admission test guarantees that the processing step will succeed. The software limit arises from the structure of our current implementation, which uses a preallocated working store and processes each scheduled virtual circuit as a separate flow in the recursively balanced scheduling algorithm. Preallocating the working store guarantees enough space to handle a certain number of flows. In Section 8.2 we discuss how we might aggregate flows to eliminate this software limit.

Once a batch of requests is admitted, it is processed by constructing a new list of scheduled virtual circuits (the bandwidth request set), running the recursively balanced scheduling algorithm on it, and installing the resulting, "new" schedule in the switch hardware. For difficult cases with many scheduled circuits, the scheduling algorithm can take a long time: up to a little more than a second in our current implementation. The switch hardware continues to forward cells according to the "old" schedule until the new schedule is complete, at which point the hardware flips from old to new at the next frame boundary. Processing requests in batches allows the software to support a high rate of circuit add/delete requests, although possibly at a latency of a couple of seconds. There is no difficulty in handling large batches because the recursively balanced scheduling algorithm always recomputes the entire schedule from scratch.

7.3. Implementation overview

The recursively balanced scheduling algorithm works by recursively splitting the problem in half, dividing the frame into two subframes and the bandwidth request set into two corresponding sets of subrequests at each level. We implement the algorithm directly using a recursive subroutine. The implementation uses three main data structures: a *schedule array* indexed by input port and frame slot, a *bandwidth request array*, and a *request-matching table*.

The schedule array constitutes the result of the scheduling algorithm. As shown in Figure 9, it is a two-dimensional array whose rows correspond to input ports, whose columns correspond to slots in the scheduling frame, and whose entries are small integers (virtual circuit indexes) identifying flows. Initially all elements in this array are set to zero, which represents an empty schedule slot. When the recursive subroutine reaches a subframe containing one slot, it fills in the slot in the schedule array based on the surviving subrequests. Because the initial problem is feasible and each step divides the problem into two feasible subproblems, we know that the final one-slot problem is feasible, which means that the surviving subrequests have a bandwidth of one cell each and no conflicts in the input or output ports. Subframes in the schedule are identified by starting slot index and count.

The bandwidth request array is organized as stack of request sets. At each recursive splitting step, two request sets are added to the stack. See Figure 10. This array is initialized with the set of input requests and provides working storage for the splitting of one set into two sets of subrequests. A set of requests is identified by its starting index and count.

The request-matching table maps integer port or link values to requests; it is used during the splitting step to find partners for odd-bandwidth requests. Actually there are two tables, one indexed by input port and one by output link. Initially these tables are empty, represented by setting all elements to the impossible index value –1. During the splitting step the requests are scanned and each request with an odd bandwidth has its index recorded in the matching tables according to its input port and output link. A match is discovered if an entry was present already in the matching table, in which case the entry is removed and the request and its match are linked together. Linking is accomplished using two extra fields per request, one for the input partner and one for the output partner. At the beginning of the splitting step, the partner fields are initialized to the impossible index value –1, again representing empty. When matches are discovered, the matching requests are linked to each other using the input or output partner field, whichever applies.

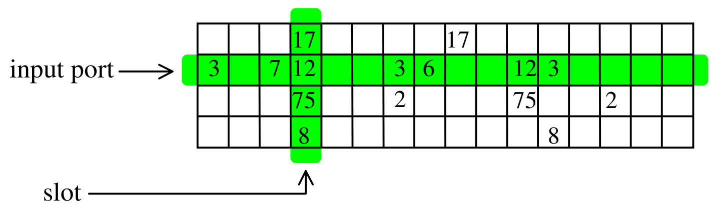

Figure 9: The schedule array.

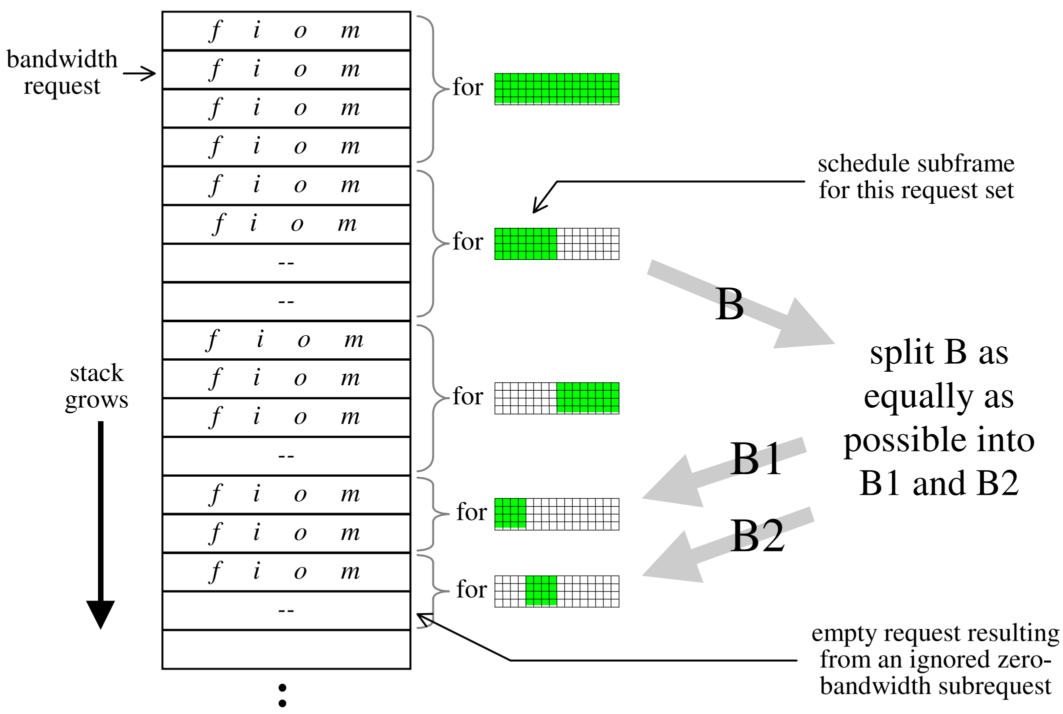

Figure 10: The array of bandwidth requests.

We need partner fields for the current set of requests only during the splitting step. To save storage space, we use a parallel array of partner fields indexed by relative request index within the current set. The parallel array is part of the working storage of the splitting step.

After all matches are discovered, the odd-bandwidth requests are split as equally as possible with respect to input port and output link by walking the cycles and paths of matched requests, favoring each subframe in alternation. At each end of a path, there will be a request with a surviving entry in a matching table. We remove this entry as we walk the path. This results in restoring the matching tables to their initial empty condition. Then we call the recursive subroutine to process each set of subrequests in turn. Because the matching tables have been restored to their initial condition, the recursive subroutine does not have to reinitialize them on each call. In the worst case, restoring the table takes time proportional to the number of requests. At the deepest levels of recursion, there are likely to be very few requests, especially if the total schedule load is not heavy. Even in the worst case, the number of requests cannot exceed the number of input ports (sixteen) times the subframe size. Because most of the invocations of the recursive subroutine occur at the deepest levels, optimizations that improve the performance at the deeper levels have large benefit.

7.4. Implementation details

Preliminary processing takes the list of scheduled virtual circuits and transcribes them into the array of bandwidth requests. This creates the initial set of requests used as input by the top-level call of the recursive subroutine. The preliminary processing also initializes the schedule array and request-matching tables.

Our implementation handles broadcast requests as well as ordinary unicast requests. Because broadcast requests use all output ports, any slot in the schedule that contains a broadcast can contain nothing else. Such a slot is called a *broadcast slot*.

Arguments to the recursive subroutine provide the starting index and count of a subrequest set in the bandwidth request array, the starting index and count of a subframe in the schedule array, and the index of the first unused entry in the bandwidth request array. We use this last index to allocate space for the two sets of subrequests, calculating the size based on the worst-case assumption that each subrequest set could be as big as the input request set.

First the recursive subroutine decides how to split its frame into subframes. If the frame size is odd, the lower subframe gets the extra slot.

Next the subroutine splits the broadcast requests. We always place broadcast requests at the front of a request set, so that we can process them first and then ignore them for the rest of the split. We split broadcast requests by scanning them in order and writing appropriate subrequests into the subrequest sets. Even-bandwidth requests split evenly into subrequests; odd-bandwidth requests split favoring each subframe in alternation. If the number of odd-bandwidth broadcast requests is odd, the lower subframe gets the extra slot of load. Totaling up the broadcast bandwidth gives the number of broadcast slots in each subframe, from which the number of remaining free slots can be computed.

When splitting odd-bandwidth broadcast requests, we don't attempt to divide each input port's load as equally as possible. Actually, this was an oversight in our implementation, because we could have formed pairs of requests sharing the same input port just as in the splitting step for unicast requests. The result of our oversight is that the load on an input port is not necessarily spread as evenly as we could manage. However, this has no effect on the feasibility of the schedule, because each unit load of a broadcast request must be scheduled into a slot all by itself. Which broadcast slot contains which broadcast unit load is immaterial to the scheduling of unicast requests.

In the case that the free slots are divided evenly between the subframes, we can proceed to split the unicast requests. Otherwise, one subframe has an *excess* free slot. In this case we first extract enough requests into the excess free slot so that the remaining requests are feasible for the remaining free slots. (The slot extraction algorithm is moderately complicated in its own right and is described in Section 7.5.) Now the remaining free slots constitute an even division of free slots between the subframes (we adjust the relevant subframe to exclude the excess free slot) and we proceed to split the unicast requests.

Once the free slots are evenly divided between the subframes, the subroutine splits the unicast requests. It takes three scans through the unicast requests to perform the split. In the first scan, requests with odd bandwidths are paired up by matching on input ports and output links. In the second scan, odd-bandwidth requests with unmatched output links are removed from the output link matching table and paired up by matching on output ports. In the third scan, cycles and paths of odd-bandwidth requests are split into the subrequest sets with each subframe favored in alternation. During one of the scans, requests with even bandwidth are split evenly into the two subrequest sets; it does not matter which scan does this.

As an optimization, we ignore all subrequests of bandwidth zero. This simple optimization turns out to save a lot of time, because all deeper levels of recursion never see the ignored subrequests.

The SPLIT properties require that the odd-bandwidth requests on each path favor each subframe in alternation, but there is no requirement that the total load be split as evenly as possible. For example, several odd-length paths could each (on balance) favor the same subframe. However, it is easy enough to keep the alternation going throughout the third scan, which results in splitting the total load as evenly as possible.[1]

After splitting all the requests, the splitting subroutine calls itself recursively to process each of the two subproblems.

7.5. Slot extraction algorithm

The method we use to extract requests into the excess free slot is a variation of the Slepian-Duguid algorithm [9, p. 66]. The Slepian-Duguid algorithm produces a complete,

[1] This would be a useful property for a switch that had a bandwidth limitation in its switching fabric, such as might be the case in a switch that used a high-speed bus for its "crossbar". With appropriate load limits on its ports, such a switch might depend on the recursively balanced splitting of total load to provide a schedule that respected its limited switching bandwidth.

legal schedule of any given frame size from any feasible set B of unicast requests. However, we need only one slot from such a legal schedule—the remaining slots we intend to compute via recursive smooth scheduling. So we divide the complete schedule into two parts: the *SLOT* and the *REST*. The SLOT is the one slot we need to extract; the REST represents the rest of the schedule.

The extraction algorithm uses three data structures: input and output port connection arrays for the SLOT; input, output, and pair load arrays for the REST; and working storage for the exchange step to be described below.

The port connection arrays give the assigned connections between input port and output port in the SLOT. A slot in a legal schedule must not have output port conflicts, so we must consider output ports and not output links when extracting the excess free slot. If it has been decided to place in the SLOT one cell of a bandwidth request b that sends cells from input port $b.i$ to output port $b.o$, then element $b.i$ in the input port connection array contains $b.o$, and element $b.o$ in the output port connection array contains $b.i$. The port connection arrays record the fact that some such b exists; there is no record of exactly which b it was. A postprocessing step will find one such b and place one of its cells in the SLOT.

The input, output, and pair load arrays give the assigned bandwidth load in the REST for each input port, each output port, and each ordered pair of input and output port.

As in Slepian-Duguid, we start with an empty schedule and feed the requests in one by one. The empty schedule is represented by an impossible value (–1) in all entries of both port connection arrays and zero in all entries of the three load arrays. As each request b is examined, we check the load arrays to see if the entire bandwidth $b.m$ would feasibly fit in the REST. If so, we increase the relevant loads by $b.m$. Otherwise, we increase the relevant entry in each of the three load arrays by $b.m-1$ and then explore how to fit the remaining cell from b into the SLOT. Assuming the request set B is feasible, $b.m-1$ will always fit into the REST.

If the request's input and output ports $b.i$ and $b.o$ are both unconnected in the SLOT, then we merely record the connection in the port connection arrays and we are done with this request. Otherwise, assuming the request set B is feasible, the only remaining possibility is that one of $b.i$ or $b.o$ is connected and the other is unconnected in the SLOT. In this case we have to do an exchange step.

The purpose of the exchange step is to reallocate traffic between the SLOT and the REST so that one cell from the request b will fit into the SLOT. This is the basic idea of Slepian-Duguid, except that here we are treating the REST as an abstract entity without caring how the detailed schedule within it might be worked out. Without loss of generality, let us assume that the request's input port $b.i$ is unconnected in the SLOT. Then the request's output port $b.o$ must be connected to some input port that we shall call $b.i_1$. We know that $b.i_1 \neq b.i$ because $b.i$ is unconnected. Now we consider *move number one*, the possibility of moving one cell of $b.i_1 \rightarrow b.o$ from the SLOT to the REST. This can be evaluated by checking the loads for $b.i_1$ and $b.o$ in the REST. If it is possible, then we move it, adjusting the loads and port connections, and then we connect $b.i$ and $b.o$ and we are done with request b.

Otherwise, move number one must be impossible because either $b.i_1$ or $b.o$ or both are completely full in the REST. It cannot be the case that $b.o$ is full in the REST, because if so the request set B would not be feasible. Hence, $b.i_1$ must be full in the REST. By consulting the pair load array, we can determine what output ports $b.o_{1a}$, $b.o_{1b}$,…, have cells from $b.i_1$ in the REST. We need to consider *move number two*, the possibility of moving one of these cells back into the SLOT. If we could perform move number two, then move number one would be enabled and then we could connect $b.i$ and $b.o$ and we would be done with request b.

Of course, in order to perform move number two, we might have to consider a *move number three*, and so on. The feasibility of the request set B guarantees that eventually we will find a move that we can perform. Then each of the earlier moves can be performed one by one back to move number one, and we will be done with request b. This assumes that we choose the correct output port in each even-numbered move.

Unfortunately, we don't know which output port to choose. Our solution is to explore them all via breadth-first search. Working storage is needed for the exploration queue and for back pointers so that when we finally find a possible move we can work our way back through all of the enabled moves to the desired state. The back pointers also serve as flags to limit later stages of the search by suppressing the exploration of output ports that are already being explored. The feasibility of the request set B guarantees that the search will terminate with success. Because the search explores each output port at most once, a reasonable bound on the search time is obtained.

It sounds horrible, having a breadth-first search down in the inner loop inside the division step of a recursive subroutine, but in practice the implementation spends almost no time there.

After we have processed all the requests, we scan them again, looking for requests whose input and output ports connect to each other in the SLOT. When we find such a request b, we write its virtual circuit index $b.f$ into the excess free slot's schedule entry for input port $b.i$ and we decrement its bandwidth by one by replacing $b.m$ with $b.m-1$. We then remove the connection $b.i \rightarrow b.o$ from the SLOT so that we won't attempt to match it again as we continue to scan the requests.

The result is a schedule for the excess free slot and a remaining bandwidth request set that is feasible for the remaining free slots.

7.6. Performance

The smooth scheduling algorithm runs on the AN2 switch's master control processor, which is a 25MHz LR33000 CPU with 8 Megabytes of memory. We have investigated the CPU time required by the smooth scheduling algorithm for various sets of requests.

Instead of running lots of experiments on the actual hardware, it was much easier to run them on a DECstation R3000 workstation of the same instruction set and approximately same speed. We ran a few experiments on the actual hardware and found that it was about five percent slower than the DECstation. The difference is probably due to the LR33000's smaller cache.

Each experimental trial consisted of a randomly generated feasible set of requests. The requests used all sixteen input and output ports, and all traffic from a given input port to a given output port was on one circuit. The experiments assumed that each output port had an output link that was capable of handling the full output port bandwidth. About one percent of the input load was broadcast traffic, and all broadcast traffic from a given input port was also on one circuit. During a run of trials, the switch output load (which was higher than the input load, because of broadcast traffic) was the same for each trial. The amount of work the smooth scheduling algorithm performs depends roughly on the amount of traffic in the schedule. Each run contained 100 trials.

Limiting the traffic to only one circuit emulates the effect of flow aggregation optimization discussed later in Section 8.2.

The results from the experimental runs are shown in Table 10.

output load	run time (milliseconds) min	average	max	worst output port discrepancy	average exchange steps
5%	273	368	477	4.94	0
10%	316	385	516	4.59	0
20%	426	481	609	4.95	0
30%	508	583	676	5.62	0
40%	609	671	750	5.54	0
50%	672	750	820	5.37	1
60%	746	825	988	5.97	42
70%	816	896	960	6.76	295
80%	871	973	1082	6.62	704
90%	937	1062	1195	6.69	1135
95%	1031	1120	1293	5.97	1454
100%	1082	1173	1266	0.00	1702

Table 10: Experimental results of the implementation of recursively balanced scheduling.

The "average exchange steps" column lists the average number of times the slot extraction algorithm had to perform an exchange step. In all the trials with less than 50% output load, no exchange steps were required.

The "worst output port discrepancy" is the worst observed scheduling discrepancy (in cells) over any interval for any output port for any trial in the run. The discrepancy falls off at very high loads because the output ports are running at almost full bandwidth—at 100% load the worst discrepancy is zero because all output ports are busy all the time. None of the experimental trials produced a discrepancy greater than seven cells, which fits comfortably within the ten cell buffers allocated to scheduled traffic in the output link's speed matching FIFOs. This is nice to see, because the AN2 switch hardware depends on the schedule being smooth enough so that these FIFOs do not overflow.

Performance could be improved by compiling the program with optimization turned on. Currently, we are more interested in debugging so we don't use optimization. In our operational AN2 network, the observed output loads are five percent or less (usually much less), and the performance of the recursively balanced scheduling algorithm is satisfactory.

7.7. An additional constraint

After we developed and implemented the recursively balanced scheduling algorithm, we discovered an additional hardware constraint. It turns out that the AN2 input/output port matcher (see Figure 1) cannot handle the case in which the same flow appears in two consecutive slots in the schedule. There is a pipeline delay, and if the last cell in the input buffer for the subject flow is sent during the first slot, the matcher will not notice in time to prevent itself from grabbing at air during the second slot, fatally corrupting the hardware's internal data structures. This problem applies to a scheduled (CBR) flow only, as the hardware specifically prevents an opportunistic (ABR) flow from sending cells in two consecutive slots.

There is a known hardware fix, but since we have not had the need to support a high volume of scheduled traffic at our laboratory, we have adopted the following draconian solution. Considering the odd slots as schedule A and the even slots as schedule B, we submit requests to schedule A only, leaving schedule B empty. Thus we support a maximum load of CBR traffic equal to half of the switch bandwidth. An improvement would take requests not admitted to schedule A and submit them to schedule B. In any case, all slots remain available for ABR traffic.

As illustrated by our earlier example in Table 7, it is not always possible to construct a legal schedule in which no flow is scheduled into successive slots, even if the bandwidth of each flow is at most half the frame size. An open question is what limitations on port loads and flow bandwidths would be sufficient to guarantee the existence of a smooth schedule in which no flow uses two consecutive slots.

8. Potential improvements

While we have so far found the performance of our implementation to be adequate, there are a number of opportunities for further performance improvements. We describe some of them here.

8.1. Parallelism

Once the bandwidth request set for a frame has been split, the scheduling tasks for the subframes are completely independent. This offers an obvious opportunity for parallelism, and since each line card in the AN2 switch has its own line card processor (LCP), there are processors available to take advantage of this opportunity. One possible organization of the computation would be for the master LCP to start by performing the top-level split and communicating one of the two resulting subrequest sets to another LCP.

Additional processors would be enlisted for lower-level splits until all processors were in use. Alternatively, all processors could start with a copy of the initial bandwidth request set, perform identical computations of the top-level split, and then continue working down the recursion tree, each restricting its attention to a designated portion of the frame and discarding subtasks not relevant to that portion. This second plan involves duplicated communication, but saves communication on the critical path, since the multiple copies of the top-level bandwidth request set can be maintained incrementally.

8.2. Aggregating flows

Our scheduling computation has three goals: (1) to connect each input port to each output port in sufficiently many slots to serve all the flows (while avoiding conflicts), (2) to conserve output buffering (and reduce output latency) by smoothing traffic to each output link, and (3) to reduce input latency by achieving smoothness on a per-flow basis. We can potentially improve performance by aggregating flows, which requires doing scheduling in two passes.

The first pass groups all flows from a given input port to a given output link as a single aggregate flow whose bandwidth request is the sum of its constituents. This pass performs the same computation as our current implementation, but treats the aggregates as we now treat individual flows. Considerable savings in time will result if the average number of flows per aggregate is large. Also, the number of aggregate flows can be no more than the product of the number of input ports and the number of output links, which is $16 \times 64 = 1024$. Hence the working stack of bandwidth request sets that is stored in the array of bandwidth requests during the smooth scheduling algorithm has a maximum worst-case size that is feasible to allocate in physical memory. Consequently, the number of flows need no longer be limited by the working stack size.

In the second pass, the slots allocated to each aggregate flow would be divided up among the individual flows, achieving per-flow smoothness. In this pass, we must consider individual flows, but the computation required is much simpler than full-blown recursive scheduling. Also, the second pass is very amenable to parallel implementation, since each LCP can process only those flows that use the switch input port connected to that LCP's line card.

8.3. Shift in strategy for small subframes

Our current scheduling implementation uses the same splitting technique for subframes of size two as for frames of size 1024. If per-link smoothness, or even per-flow smoothness, is abandoned within subframes of small size—say 16 or fewer slots—the consequent loss of smoothness may be small enough not to be of concern. In this situation, scheduling within these small subframes may be performed by a more efficient technique than our recursively balanced scheduling algorithm. Efficiency improvements in small subframes can have a large impact on the performance of the entire algorithm, because most of the work occurs at the deepest levels of recursion, which deal with small subframes.

Under loads that are not extremely heavy, such a shift in strategy opens up the possibility for efficient *incremental updates* to the schedule. For example, if the total bandwidth request set loads no port to more than 15/16 capacity, then after splitting the request set down to subframes of size 16, each subframe will have a subrequest set that is actually feasible for 15 slots. If these subrequest sets are each scheduled into 15 slots (possibly without regard to smoothness), then one slot in 16 will be left completely free of scheduled traffic. We could then schedule small new requests incrementally into those slots, and periodically run the full-blown recursively balanced scheduling algorithm in the background to repack the schedule. Leaving some slots free, or nearly free, of scheduled traffic also serves to improve the performance of opportunistic traffic by increasing the probability that an opportunistic flow's input and output ports will both be available in the same time slot. This effect is illustrated in Figure 8.

The idea of incrementally updating a schedule instead of recomputing it from scratch had also been applied to satellite switching TDMA networks [2].

9. Conclusions

The recursively balanced scheduling algorithm improves switch latency and buffering requirements for scheduled traffic by smoothing out the workload of each flow and output link. The algorithm produces smooth schedules all the way up to 100% switch load. Previous work provides no guarantee of smoothness when scheduling heavy load. We implemented the algorithm as part of the switch control software in AN2, which has been in daily use as a service network at our laboratory since the end of 1994. Because we knew we could produce smooth schedules, the AN2 quad line card includes only a small speed-matching FIFO for each output link. Without the recursively balanced scheduling algorithm, this design would not work. Performance of the implementation has been satisfactory.

Acknowledgments

Allan Heydon, Cynthia Hibbard, and Michael Schroeder provided helpful comments and suggestions on earlier drafts of this report.

References

1 Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High speed switch scheduling for local area networks. *ACM Transactions on Computer Systems*, 11(4):319-352, November 1993. Also available as Research Report SRC-99, Digital Equipment Corporation Systems Research Center, April 1993.

2 Wen-Tsuen Chen and Huai-Jen Liu. An adaptive scheduling algorithm for TDM switching systems. *IEEE Transactions on Communications*, 43(2/3/4):651-658, February/March/April 1995.

3 Wen-Tsuen Chen, Pi-Rong Sheu, and Jiunn-Hwa Yu. Time slot assignment in TDM multicast switching systems. In *Proceedings, Tenth Annual Joint Conference of the IEEE Computer and Communications Societies, Volume 3*, pages 1296-1305, 1991.

4 Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, April 1993.

5 Joseph B. Evans, Douglas Niehaus, David W. Petr, Victor S. Frost, Gary J. Minden, and Benjamin J. Ewy. A 622 Mb/s LAN/WAN gateway and experiences with wide area ATM networking. *IEEE Network*, 10(3):40-48, May 1996.

6 S. Jamaloddin Golestani. Congestion-free transmission of real-time traffic in packet networks. In *Proceedings, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies, Volume 2*, pages 527-536, 1990.

7 S. Jamaloddin Golestani. A framing strategy for congestion management. *IEEE Journal on Selected Areas in Communications*, 9(7):1064-1077. September 1991.

8 Inder S. Gopal and C. K. Wong. Minimizing the number of switching in an SS/TDMA system. *IEEE Transactions on Communications*, COM-33(6):497-501, June 1985.

9 Joseph Y. Hui. *Switching and Traffic Theory for Integrated Broadband Networks*. Kluwer, Boston, 1990.

10 Joseph Y. Hui and Edward Arthurs. A broadband packet switch for integrated transport. *IEEE Journal on Selected Areas in Communications*, SAC-5(8):1264-1273, October 1987.

11 Mark J. Karol, Michael G. Hluchyj, and Samuel O. Morgan. Input versus output queuing on a space-division packet switch. *IEEE Transactions on Communications*, COM-35(12):1347-1356, December 1987.

12 Manolis Katevenis, Panagiota Vatsolaki, and Aristides Efthymiou. Pipelined memory shared buffer for VLSI switches. *ACM SIGCOMM Computer Communication Review*, 25(4):39-48, October 1995. From Proceedings of ACM SIGCOMM'95.

13 C. A. Pomalaza-Raez. A note on efficient SS/TDMA assignment algorithms. *IEEE Transactions on Communications*, 36(9):1078-1082, 1988.

14 Thomas L. Rodeheffer and Michael D. Schroeder. LAN emulation using resilient virtual circuits. In preparation.

15 Charles P. Thacker. Method and apparatus for resource arbitration. U.S. Patent 5,267,235, November 1993.

16 Charles P. Thacker and Michael D. Schroeder. AN2: A high-performance ATM switch. In preparation.

17 Yiu Kwok Tham. On fast algorithms for TDM switching assignments in terrestrial and satellite networks. *IEEE Transactions on Communications*, 43(8):2399-2404, August 1995.

18 Fouad A. Tobagi. Fast packet switch architectures for broadband integrated services digital networks. *Proceedings of the IEEE*, 78(1):133-167, January 1990.