Fair Scheduling in an Optical Interconnection Network

Ch'ng Shi Baw
Roger D. Chamberlain
Mark A. Franklin

Ch'ng Shi Baw, Roger D. Chamberlain, and Mark A. Franklin, “Fair Scheduling in an Optical Interconnection Network,” in *Proc. of 7th Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, October 1999, pp. 56-65.

Computer and Communications Research Center
Washington University
Campus Box 1115
One Brookings Dr.
St. Louis, MO 63130-4899

Fair Scheduling in an Optical Interconnection Network *

Ch'ng Shi Baw baw@ccrc.wustl.edu
Roger D. Chamberlain roger@ccrc.wustl.edu
Mark A. Franklin jbf@ccrc.wustl.edu
Computer and Communications Research Center
Washington University, St. Louis, MO

Abstract

Existing fair scheduling schemes have focused primarily on scheduling multiple flows to a single output. The limited work that has focused on scheduling multiple flows to multiple outputs has assumed a non-blocking, slotted-time, cell-based network with a centralized controller. This paper presents a fair scheduler suitable for use in bufferless circuit-switched blocking networks operating with distributed, asynchronous controllers and variable length messages. We begin by describing the potential for starvation in the Gemini interconnect network, an optical, circuit-switched network. A proposed distributed fair scheduler is presented and shown to solve this problem. The tradeoffs and limitations of performing many-to-many fair scheduling in general, and that of our fair scheduler in particular, are discussed.

1 Introduction

Traditional fair scheduling techniques have focused on scheduling flows at the outputs of switching systems [6, 7, 11]. These schedulers necessarily assume an output queued system. Switching systems are thus expected to provide higher internal bandwidth (relative to external link bandwidth) to present to the schedulers the abstraction of an output queued system. In today's switching systems, the ratio of internal to external bandwidth (referred to as *speedup* or *speed advantage*) is usually greater than one. The scheduler at each output executes independently of each other and focuses on scheduling the flows at the output to one external link. Thus traditional fair schedulers support only many-to-one scheduling.

A few works that explicitly support many-to-many scheduling (multiple inputs sending to multiple outputs in parallel) have thus far been limited to using a centralized controller and a non-blocking switch fabric. In particular, Chuang et al., in an effort to devise a many-to-many scheduler that can emulate an output queued system (where fair scheduling is assumed be done at each output port independently), has shown in [5] that to faithfully emulate an output queued system (assuming an $N \times N$ non-blocking switch fabric), the speed advantage needs to be $(2 - \frac{1}{N})$ or greater. Chuang's work assumes a cell-switched environment where switching is done using conventional electronic technology.

In our study of the Gemini interconnect architecture (targeted at Massively Parallel Processing (MPP) systems), we have found an instance of a fair scheduling problem where all the assumptions discussed above (and a few others to be discussed later) are violated.

The Gemini network, first proposed in [4], combines an optically switched network and a conventional electrically switched network to provide a high bandwidth interconnect targeting the class of *space-time adaptive processing (STAP)* applications. STAP applications require passing of large data blocks among processors as well as short synchronization messages. Furthermore, the performance of these applications is sensitive to the latency of short synchronization messages.

The optically switched network, while providing high bandwidth, requires path setup prior to transmission. Such overhead adds latency. The electrically switched network, being packet switched, does not need path setup and hence does not incur this overhead. However, the electrically switched network is limited in terms of bandwidth. The Gemini network attempts to take the best of both worlds by sending long data messages over the optical network and sending short messages through the electrical network.

Sending messages through an electrical interconnection network is a well studied subject and hence will not be considered here. An efficient optical network control protocol has also been developed for use in the Gemini network and has been reported in [1, 2]. The focus of this paper is on starvation prevention and fair scheduling issues in the optical network. A simulation tool, the *Interconnection Network Simulator (ICNS)* whose design and implementation are documented in [3], is used extensively to study this problem. All simulation results presented here were generated using ICNS.

Section 2 overviews the Gemini architecture and the

*Work reported herein was supported in part by NFS grant EIA-9706918.

basic optical network control protocol. Section 3 describes the starvation issue that motivated our effort in many-to-many fair scheduling. Section 4 explains the concept of fairness as it pertains to many-to-many scheduling. Section 5 describes the *Distributed Deficit Round Robin (dDRR)* scheduler developed for use with Gemini. Section 6 presents complexity analysis and simulation results of the dDRR scheduler. Section 7 concludes this paper.

2 The Gemini Network

2.1 Topology and Architecture

The Gemini network attempts to perform optical switching without using optical amplifiers within the interconnection network [2, 4]. Optical switching elements attenuate the incoming signal as they switch the signal [8]. Thus to minimize switch insertion loss, the Gemini network uses a Banyan topology to minimize the number of switches through which signals have to pass. The electrical network is topologically identical to the optical network. An 8×8 Gemini network is shown in Figure 1. As can be seen in Figure 1, each optical switch has its electrical counterpart.

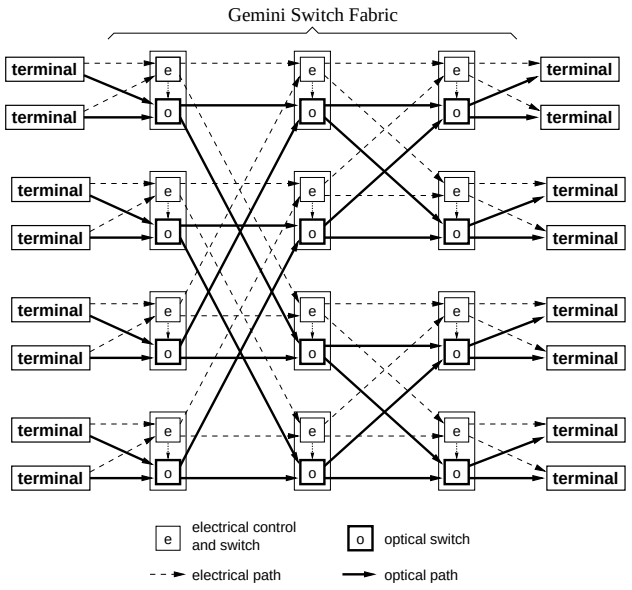

Figure 1: An 8×8 Gemini network.

The optical switching elements used in the Gemini network are $LiNbO_3$ based 2×2 switches. An optical switch has two states, the $\times$ state and the $=$ state as shown in Figure 2. The optical switch is passive and has no buffering capability. These switches rely on the electrooptic effect (i.e., the application of an electric field changes the refractive index of a material within the field) to provide for pass through and crossover connections between the input and output ports [8]. The bias voltage is controlled electronically by the optical switch's electrical counterpart. The optical network is circuit switched and unidirectional.

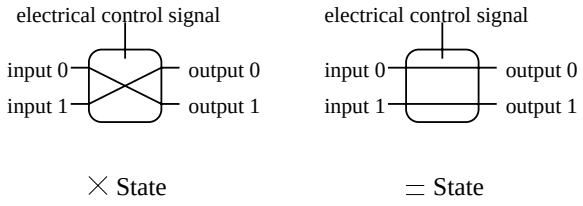

Figure 2: The two states of a Gemini optical switching element.

The Gemini optical network assumes no centralized controller. Thus network control signals need to be passed from switch to switch using the electrical network. The electrical network is packet switched. Packet switching allows flexible intermixing of user data and control signals in the electrical network. The electrical switch has a shared input buffer and separate output buffer at each output. Each electrical switch controls its companion optical switch.

2.2 The VOQ Protocol in Gemini

This section describes the underlying protocol used to control Gemini's optical network. The control signals of interest are: *setup*, *ackSetup*, *block*, and *teardown*. The basic protocol is as follows:

Suppose a terminal S wants to send a long message to another terminal D. An optical path needs to be setup so that light pulses transmitted by S's laser output can be detected by D's detector. Suppose there are three stages of switches in the network as shown in Figure 3.

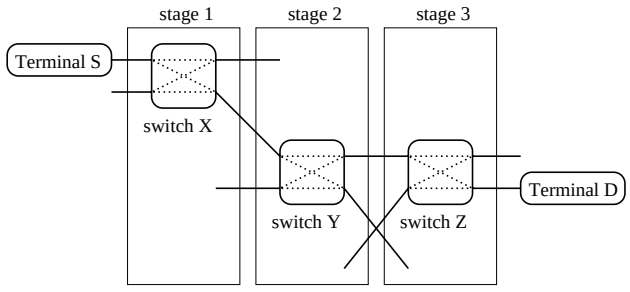

Figure 3: Terminals S and D connected via a 3-stage network.

To setup an optical path, S sends a *setup* signal to D. The *setup* signal has to pass through switches X,

Y, and Z before it reaches D. If X, Y, and Z all grant S's *setup* request, D will send a *ackSetup* signal to S. S can start optical transmission after it receives D's *ackSetup* signal. If one of the switches, X, Y, or Z, cannot grant S's setup request, the switch will set a *blocked* flag in the *setup* signal. If D receives a *setup* signal with the *blocked* flag set, D will send S a *block* signal back to S. S will send a *teardown* signal to D if it receives a *block* signal or once it has finished its optical transmission.

A switch grants a *setup* request if the switch is not already reserved in a state conflicting with that asked for by the *setup* request, and blocks the request otherwise. If a switch grants a *setup* request, it immediately assumes the state asked for by the request.

The Banyan topology has poor blocking characteristics. Thus instead of using traditional FIFO queues at the input terminals, we use *virtual output queues (VOQ)* to avoid head-of-line blocking. Using VOQ, outgoing messages at input terminals are queued according to their output destinations.

Using VOQ, each input terminal sends a *setup* request for each of its non-empty virtual output queues. If all *setup* requests are blocked (i.e., the terminal recieves a *block* for each *setup* it has sent), the terminal waits some time before it sends the requests again. However, if one of the requests is granted, the input terminal will transmit a message associated with the granted request[1]. For each blocked request, the terminal sends an associated *teardown* signal.

The above is termed the *VOQ Protocol*. The throughput improvements obtained by using the VOQ protocol are reported in [1, 2]. Here, we concern ourselves with the starvation problem introduced by the VOQ protocol and a fair scheduling solution, the *Distributed Deficit Round Robin (dDRR)* scheduler, proposed to solve the problem.

3 Starvation Issues with VOQ

This section describes how starvation can occur, identifies the root cause of starvation, and shows the general trend of starvation with respect to network load in the Gemini network. Consider a simple 2×2 network. Using VOQ, each sender maintains two queues – one for each destination. As shown in Figure 4, assume all four queues are non-empty.

Suppose $S1$ successfully sets up a path to $D2$, the switch i will necessarily be in a $\times$ state until $S1$ sends a *teardown*. Since $S2$'s queues are non-empty, it will attempt to setup paths to $D1$ and $D2$ according to the VOQ protocol. Of the two setup attempts, the one

[1] In a Banyan topology, the terminal is guaranteed that at most one request can be granted.

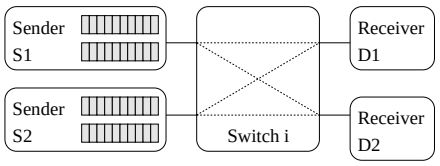

Figure 4: A 2×2 network with VOQ.

to $D1$ will be successful because i is already in the $\times$ state. Thus switch i is reinforced by $S2$ to stay in the $\times$ state.

At a later time, either $S1$ or $S2$ will finish transmission and tear down their optical path. However, it is unlikely that the two will finish transmission and begin tear down at the same time. Thus after a sender, say $S1$, finishes tear down, the other sender $S2$ will still have its optical path reserved, keeping switch i in the $\times$ state. As $S1$ begins a new path setup cycle, its setup request to $D2$ will be granted and its request to $D1$ will be blocked. $S1$ further reinforces switch i to stay in the $\times$ state. $S1$ will not have the opportunity to send to $D1$, and $S2$ will not have the opportunity to send to $D2$ as long as switch i stays in the $\times$ state.

Switch i will be repeatedly reinforced to stay in the $\times$ state until some of the queues become empty or until an unlikely event occurs – that $S1$ and $S2$ tear down their paths at the same time. Under heavy load, queues rarely become empty. Thus some queues may be denied service for a long time, thus resulting in starvation.

An alternative scenario is that $S1$ has an unlimited number of messages that it wants to send to $D2$ but nothing to send to $D1$, and $S2$ has data to send to $D2$. Suppose $S1$ succeeded in setting up a path to $D2$, then following similar reasoning described above, we see that $S2$ may never get to send data to $D2$. Thus starvation may occur even if only one queue is consistently non-empty.

3.1 The Effect of Network Load on Starvation

Figures 5, 6, and 7 show the effect of VOQ induced starvation in a 4×4 network via simulation. There are $4^2 = 16$ possible flows in the network. We number the flows 1 to 16. The network load is generated using a Poisson process with exponentially distributed message lengths. Destinations of messages are uniformly distributed to all outputs.

In Figure 5, we plot the cumulative amount of traffic sent for each flow at different times. The input load is 0.8. Each horizontal dotted line connects a set of cumulative traffic measurements for each flow taken at a given point in simulation time (hereafter referred to

as a *snapshot*). As we move up the graph, time increases, and the total number of bits transmitted increases. The time elapsed between successive snapshots is roughly equivalent to the time needed for a sender to send about 500 average length messages ($\Delta t = 1.0$ in simulation) in a contentionless environment. Pictorially, we see that the dotted lines are roughly flat. This indicates that each flow has roughly the same amount of access between snapshots.

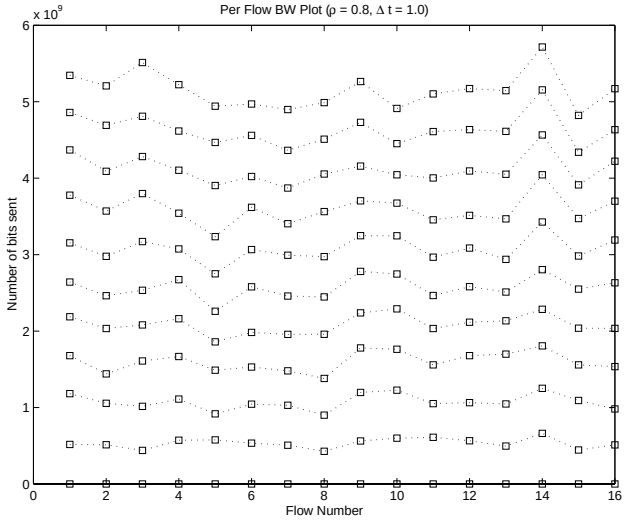

Figure 5: Access amount snapshots at 500 message time interval at 0.8 load.

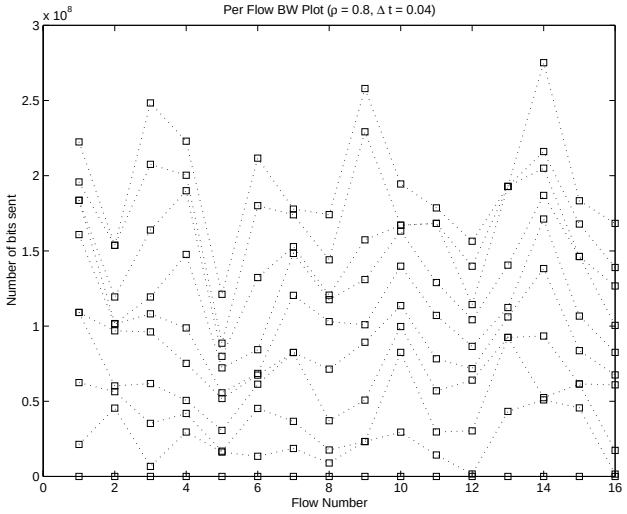

Figure 6: Access amount snapshots at 20 message time interval at 0.8 load.

Figure 6 is the time scale *zoom in* of Figure 5. The elapsed time between snapshots is reduced to roughly

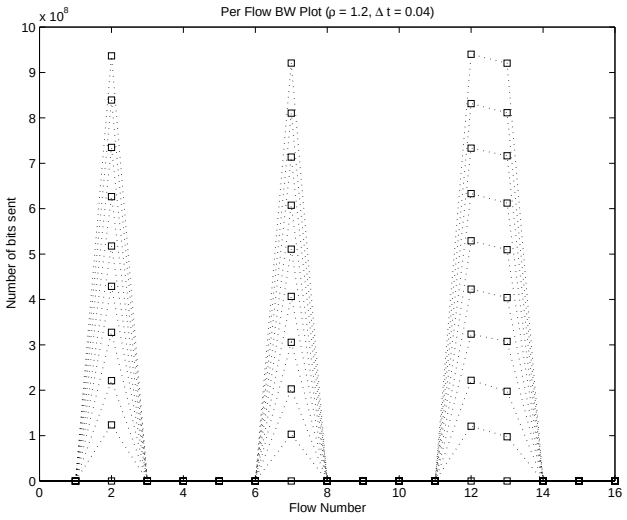

Figure 7: Access amount snapshots at 20 message time interval at 1.2 load.

20 message times ($\Delta t = 0.04$). On the smaller time scale, we see that not all flows are serviced by the same amount at each interval. Part of the reason is that some queues may be empty at times. However, all flows are serviced, and these results do not indicate starvation.

For Figure 7, the input load was increased to 1.2. The elapsed time between snapshots is roughly 20 message times ($\Delta t = 0.04$). It clearly shows that all flows except 2, 7, 12, and 13 are starved during the entire 200 message time window in which the snapshots are taken.

The results supports our intuition that during under high load, it is less likely that some queues become empty, hence it takes longer to get switches to change state, thus the more likely that starvation occurs.

3.2 The Tradeoff Between Fairness Granularity and Throughput

To avoid the *unfair* use of the optical network by the various flows, it is necessary that there be a mechanism to prevent switches from staying in the same state for a long time. The scheduler's purpose is to cause the switches to change states frequently enough that flows are not denied access for too long. Furthermore, under the variable-length message assumption where messages are not segmented into fixed sized blocks (i.e., not cell-based), the scheduler cannot simply force a switch to change state at an arbitrary time. A switch has to wait until there is no active connection going through it before it can change state. This restriction leads to the throughput versus fairness tradeoff described below.

Using the 2×2 network example shown in Figure 4, suppose switch i is in the $\times$ state, $S1$ is sending to $D2$, and $S2$ is sending to $D1$. After $S1$ has torn down its connection, if we were to insist that $S1$ sends only to $D1$ the next time, switch i will need to switch state after $S2$ tears down its connection. To do so, switch i will need to deny all requests from $S1$ until $S2$ tears down its connection. Only then can the switch grant a request that requires it to be in the $=$ state and allows $S1$ to send to $D1$.

There is a time gap in which $S1$ has data to send but cannot send the data. We compromised throughput to avoid starvation. Intuitively, we can see that the more frequently we force the switches to change state, the more we compromise throughput. Thus, the finer the time granularity we require connections to be served fairly, the more we compromise throughput. Fairness granularity versus throughput is an inherent tradeoff in a blocking circuit-switched network such as Gemini's.

4 Fair Scheduler Concept and Design Considerations

This section describes the *fairness* concept as it pertains to many-to-many contention systems and proposes a quantitative measure of fairness for such systems. Concerns that affect the design choice for the Gemini fair scheduler in particular will be addressed and, where appropriate, quantitative measures proposed to address these concerns.

Intuitively, if k parties contend for some divisible resource R and each party gets $\frac{1}{k}$ of R, then we say that the division is fair. The measure for *fairness* in terms of different flows contending for access to the optical network is similar.

Fair schedulers aim to ensure that different connections are serviced fairly when there is contention. It is a well studied subject as the large amount of literature [6, 7, 9, 11] devoted to it can attest. However, most work in fair scheduling relies on one or more of the following assumptions:

1. Different queues contending for one data link where the focus is how to determine which queue should send next [6, 7, 10, 11] (which implies that there is a centralized controller).

2. Multiple senders contend for multiple receivers where the senders and receivers are connected through a non-blocking network [9].

3. Time is slotted [9] (which implies that there is a global clock).

4. Buffers are available at switches, in which case one can queue messages for different flows in the switches and execute a fair scheduler that works under assumption 1 above at each switch [10].

The Gemini optical network, being a bufferless, circuit-switched blocking network, violates every assumption listed above.

4.1 Quantitative Measure of Fairness

To quantitatively measure fairness, we use the *FairnessMeasure* metric [11]. We modify the original definition of *FairnessMeasure* to better account for the differences between the many-to-many scheduling model such as Gemini's and the many-to-one scheduling model for which *FairnessMeasure* was originally defined.

Two flows, i and j, can form a *valid set of contending flows* if and only if i and j contend for a resource (e.g., a switch, an output, an input). If $\mathcal{F}_1$ and $\mathcal{F}_2$ are two valid sets of contending flows, then their union $\mathcal{F}_1 \cup \mathcal{F}_2$ is a valid set of contending flows if and only if there exist $i \in \mathcal{F}_1$ and $j \in \mathcal{F}_j$ such that $\{i, j\}$ is a valid set of contending flows. Nothing else is a valid set of contending flows.

When measuring fairness over a valid set of contending flows $\mathcal{F}$ over a continuous time interval $\mathcal{I}$, the measurement makes sense only if all the flows in $\mathcal{F}$ are actively contending for access to a shared resource throughout the time interval $\mathcal{I}$. In Gemini, each flow is associated with a virtual output queue at an input terminal. Each flow is said to be *actively contending* for network access during the interval $\mathcal{I}$ if and only if its associated queue is never empty during $\mathcal{I}$. To make our (modified) *FairnessMeasure* metric concrete, we define the following quantities:

- $a_i(\mathcal{I})$: the amount of access received by flow i ($i \in \mathcal{F}$) during the interval $\mathcal{I}$. For example, if flow i successfully delivers k bits over the network during the time period $\mathcal{I}$, then $a_i(\mathcal{I}) = k$ bits.

- $q_i(\mathcal{I})$: a weight, a finite positive real number, assigned to flow i, $i \in \mathcal{F}$. Defined such that $q_i(\mathcal{I}) / \sum_{j \in \mathcal{F}} q_j(\mathcal{I})$ indicates the fraction of access that flow i should *ideally* receive from the total amount of access received by all flows in $\mathcal{F}$ in the interval $\mathcal{I}$.

For simplicity, we assume that during the time period of interest, the weights assigned to the flows do not change. Thus we drop the interval argument $\mathcal{I}$ from the above quantities in future notation.

Using the above quantities, the *FairnessMeasure* for the time interval of interest $\mathcal{I}$, $FM(\mathcal{I})$, is defined as follows:

$$FM(\mathcal{I}) \equiv \max_{i,j \in \mathcal{F}} \left| \frac{a_i / \sum_{k \in \mathcal{F}} a_k}{q_i / \sum_{k \in \mathcal{F}} q_k} - \frac{a_j / \sum_{k \in \mathcal{F}} a_k}{q_j / \sum_{k \in \mathcal{F}} q_k} \right| \quad (1)$$

The metric $FM(\mathcal{I})$ in effect measures the greatest normalized descrepancy between any two flows. In a perfectly fair system, $FM(\mathcal{I}) = 0$ for any interval $\mathcal{I}$. In the least fair system where one flow monopolizes access and all the other flows starve, if all flows are assigned the same weight, then $FM(\mathcal{I}) = |\mathcal{F}|$.

For an $N \times N$ multistage switching network, there are $|\mathcal{F}| = N^2$ flows. The most unfair access pattern occurs when all the switches maintain their states throughout the entire interval of interest. During the interval, exactly N flows each consume $\frac{1}{N}$ of the network bandwidth. Neglecting signaling delays, if each flow is assigned equal weight, then $FM(\mathcal{I}) \approx N$ for sufficiently long interval $\mathcal{I}$.

4.2 Granularity of Fairness

Actual interconnection networks can only send data in units that are within certain size ranges. For example, a b-bit wide shared bus interconnect sends b bits of data in each bus cycle. If there are N devices contending to access the bus, assuming all the devices have the same priority and assigned the same weight, the best a fair arbiter can do is to allow each device to access the bus for one cycle in a round-robin manner. Clearly, the arbiter cannot enforce fairness with a resolution any finer than b bits. Furthermore, it takes at least N bus cycles to allow each of the N devices a chance to access the bus. Thus the arbiter's fairness measurement for any interval smaller than N bus cycles is meaningless.

In Gemini, each time a flow successfully contends for access, it gets to send a message. Since access is granted in terms of messages and message sizes may vary, the fairness resolution cannot be smaller than the size of the largest message.

Even when we have a modest upper bound on message size, fairness granularity as fine as the message size upper bound may not be necessary. There may be reasons to prefer coarser grained fairness. We mentioned before in Section 3.2 that there is an inherent tradeoff between the granularity of fairness and throughput in a blocking circuit-switched network. Thus we would like a scheduler that can operate at different specified granularity of fairness so that one can tune the scheduler to optimize for throughput or fairness as is appropriate depending on the applications.

4.3 Scheduler Design Considerations

A fair scheduler needs access to queue state information associated with the set of contending flows in order to properly perform fair scheduling. Furthermore, in a blocking network, the scheduler needs to be aware of network topology. In the case of Gemini, we would like to leverage the existing data and signaling paths and the VOQ protocol. We would like to integrate fair scheduling and optical path setup tightly to avoid adding too much overhead and latency to the VOQ protocol. Since the original VOQ protocol requires each flow to send *setup* signals through the relevant switches to setup an optical path, queue state information can be piggy-backed on the *setup* signal. Naturally, the *block* and *ackSetup* signals can be used to convey the results of contention as before. Seeing that we can easily aggregate and convey queue state information to the switches, we decided to implement a distributed scheduler in the switches. By so doing, topology-induced blocking can be inherently taken into account.

To keep the signaling latency low, the switch needs to track queue state information, make scheduling decisions, and decide whether to block a *setup* request quickly. It is also important to have a sense of how much cost a scheduler would add to our system. Thus we add to our metrics a measure of complexity with which to evaluate the Gemini fair scheduler. The complexity metric comes in two dimensions: space and time.

Space Complexity refers to the amount of storage needed for the scheduler to track the states of all contending queues.

Time Complexity refers to the number of sequential computational steps that the scheduler needs to perform to make one scheduling decision.

5 Distributed DRR Scheduler

We based our scheduler on the *Deficit Round Robin (DRR)* scheduler [11]. We call the adapted scheduler the *Distributed DRR (dDRR)* scheduler since it is a distributed scheduler based on DRR. Because Gemini's assumptions are very different from the original DRR's, the round-robin notion is not enforced in dDRR.

DRR's basic idea is to assign each flow a *quota* which reflects how much access the flow is allowed to have during a *round*. The quota is called the *quantum*. A quantum is effectively a weight assigned to a flow. The scheduler is said to have "visited" a flow if it checks the state of the flow for the purpose of determining whether it should allow the flow to send data. Two visits are said to be interleaved for a flow if the scheduler first visits the first flow, then visits some other flows, and

then revisit the first flow. A *round* is the interval during which every actively contending flow is visited by the scheduler at least once and no flow receives two interleaved visits. Each flow has a *deficit counter* that keeps track of how much of the flow's quota has been used up, or how much unused quota the flow has accumulated. At the beginning of each round, each deficit counter is replenished by an amount equal to its flow's quantum. To prevent a flow that is not actively contending for network access from accumulating unused quota at each round, the flow's deficit counter is reset once it stops contending[11].

The dDRR scheduler employs the same basic idea but the environment in which dDRR is expected to operate is very different from DRR's. Unlike DRR in which the scheduler is located with the queues it serves and has ready access to the queue's state information, such information needs to be made explicitly known to dDRR. Furthermore, while a DRR scheduler visits its queues one by one, *setup* requests from different flows "approach" partial dDRR schedulers in different switches in no particular order.

As mentioned before, our fair scheduler is implemented in the switches. Thus each switch will contain a (partial) dDRR scheduler. We first describe how queue state information can be passed to the switches and then describe the operation of the dDRR schedulers in the switches.

5.1 Conveying Scheduling Information to the Schedulers

A fair scheduler needs to know how much access a flow wants, and how much access a flow has consumed in order to arbitrate fairly. In dDRR, such information is conveyed to the schedulers via the *setup* and *teardown* signals.

The *setup(i,amount,blocked)* signal represents a request by flow i to send *amount* units of data. The *blocked* variable may be modified by the switches to indicate if the request has been blocked, either due to scheduling or network blocking.

The *teardown(i,amount,more)* signal represents an acknowledgement by flow i that it has successfully sent *amount* units of data. The *more* parameter indicates if flow i still has more data to send (i.e., flow i's queue is non-empty).

5.2 The dDRR Scheduler in Switches

To implement dDRR in Gemini, we need to implement a partial dDRR scheduler in each switch node. Figure 8 shows how an incoming *setup* or *teardown* signal flows through the controller.

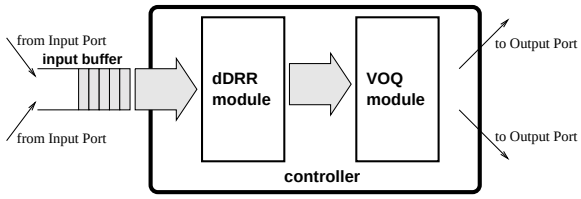

Figure 8: Structure of a dDRR Gemini switch controller.

The VOQ module processes control signals according to the VOQ Protocol described in Section 2.2. The advantage of this structure is that if the fair scheduling function is not desired for a particular application, it can be turned off and the dDRR module simply passes every packet it sees to the VOQ module untouched. This allows the dDRR module to concentrate on fair scheduling without taking blocking into account. Blocking is a concern resolved by the VOQ module.

For each flow i that goes through the switch x, the dDRR module at switch x keeps the following state information for i:

- q_i^x: flow i's *quantum*. The amount of access flow i should be granted in each *round* if flow i is actively contending. This is effectively flow i's weight. (The notion of a *round* will be made clear later.)
- dc_i^x: flow i's *deficit counter*. This counter keeps track of how much access quota flow i has consumed or has accumulated from previous rounds. This variable represents i's available quota at any given time. dc_i^x is initially set to q_i^x.
- $more_i^x$: keeps track of whether i is actively contending. This flag is set as long as i is actively contending. $more_i^x$ is initially cleared.
- sp_i^x: flow i's *suspension* flag. This flag is set if flow i requests an amount of access greater than it's available quota or if i has stopped contending. sp_i^x is initially set.
- nr_i^x: flow i's *new round* flag. This flag is set if i's *setup* signal has not been received in the current round. nr_i^x is initially set.

The state information described above is called an *entry*. Each flow has an entry in every switch it goes through. We assume that proper values are assigned to *quanta* before dDRR is activated. For any given flow, its quantum value at each switch is the same. We

drop the superscript and/or subscript when the context makes it clear which variable is being referred to.

A *round* for a switch is a time interval during which every flow that is actively contending for access at the switch either successfully consumed the maximum amount of access allowed by the flow's available quota or the flow has stopped contending because it has no more data to send. Since different switches have different sets of contending flows, *rounds* are not synchronized among switches.

When processing a signal *setup(i,amount,blocked)*, a dDRR module checks if i has been suspended before and if i is *not* in a new round. If so, it denies i's request. Otherwise, it checks if i's requested amount of access is within its available quota. If so, i is granted access. Otherwise, the dDRR module denies i's request and set its *suspension* flag. In this case, it will also replenish i's quota if this is a new round for i. Regardless of whether i's access has been granted or denied, the dDRR module remembers that i has asked for access in this round (i.e., this round is no longer a *new round* to i). If dDRR decides to *deny* a request, it sets the *blocked* bit in the *setup* signal. Otherwise, it leaves the signal untouched. The dDRR module passes the signal to the VOQ module after it has processed the signal. The pseudo code in Figure 9 describes how dDRR processes a *setup(i,amount,blocked)* signal.

```
if sp_i ∧ ¬nr_i
    set blocked_i
elseif dc_i < amount
    if nr_i
        dc_i ← dc_i + q_i
    set blocked_i
    set sp_i
else
    unset sp_i
set more_i
unset nr_i
pass setup(i,amount) to the VOQ module
```

Figure 9: Pseudo-code for processing a *setup(i, amount, blocked)* signal.

We see that processing a *setup(i, amount, blocked)* signal takes $O(1)$ time. When processing a *teardown(i,amount,more)* signal, dDRR subtracts *amount* from flow i's deficit counter if *more* is set. Otherwise, it sets i's deficit counter to equal i's quantum and sets i's *suspension* flag. The *more* argument gets copied into $more_i$. The pseudo code in Figure 10 describes how dDRR processes a *teardown(i,amount,more)* signal. Processing a *teardown(i,amount,more)* signal also takes $O(1)$ time.

```
if more
    dc_i ← dc_i − amount
else
    dc_i ← q_i
    set sp_i
more_i ← more
pass teardown(i,amount,more) to the VOQ module
```

Figure 10: Pseudo-code for processing a *teardown(i, amount, more)* signal.

Recall we defined a *round* as the period during which all flows have either used up their quota so none can send its next message or have stopped contenting. Thus a dDRR scheduler reaches a round boundary when all *suspension* flags become set. When a round boundary is reached, all *new round* flags are set and for every flow i, the *suspension* flag of i is set to the complement of the *more* flag of i. Checking the *suspension* flags and setting the *suspension* flags can be done in parallel with minimal hardware support. Thus these operations can be made to take $O(1)$ time.

6 Evaluation of dDRR

6.1 Complexity of dDRR

We have shown previously that dDRR has a time complexity of $O(1)$ for all its operations. To determine the space complexity of dDRR, we need to know how many flows go through each switch. In a Gemini $N \times N$ Banyan network, a switch at stage j can be reached by at most 2^{j+1} input terminals and the switch itself can reach at most $N/2^j$ output terminals. Thus the total number of flows that go through one switch is $2^{j+1} \cdot N/2^j = 2N$. Since each flow has one set of state variables, the space complexity for dDRR is $O(N)$ per switch, or $O(N^2 \log_2 N)$ over the entire network.

6.2 Performance of dDRR

In Figure 7, we showed that different flows are given different amounts of access using the original VOQ protocol. Using the same parameters, we rerun the simulation with dDRR. Each flow is assigned a quantum that is eight times the average message size. Figure 11 shows the results, and demonstrates that starvation has been eradicated.

Because we placed no restriction on message size and did not ensure that all flows are actively contending

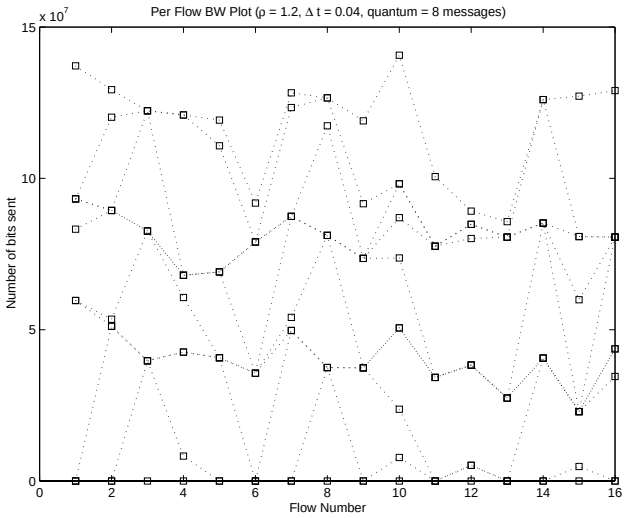

Figure 11: Service amount snapshots at 20 message time interval at 1.2 load.

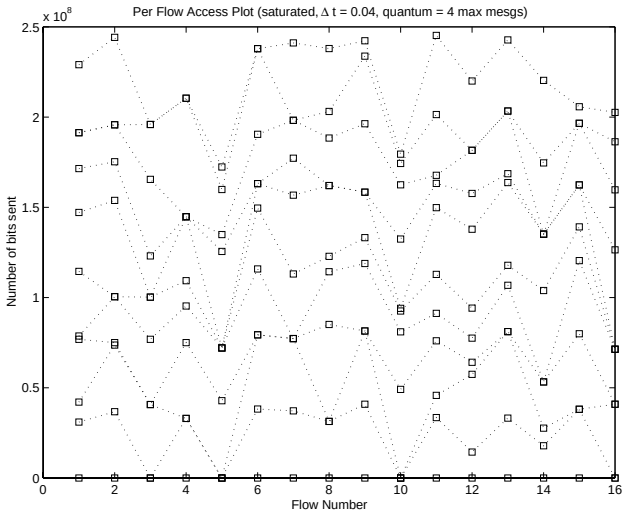

Figure 12: Service amount snapshots at 20 message time interval. Saturated queues, quantum = 4 MAX.

throughout the previous simulations, we cannot meaningfully comment on how fairly dDRR has distributed access to various flows. To investigate how fair dDRR performs, another set of simulations were performed. This set of simulations used the same parameters as previous simulations except all queues were presaturated with messages and the message size was restricted to be no larger than MAX.

Figure 12 shows the amount of service different flows have consumed during an interval of 0.5 sec simulation time. All quanta were set to four times the maximum message size (i.e., quantum = 4 MAX). A *FairnessMeasure* of 0.33 was achieved.

Figure 13 shows the results for a simulation where the quanta were set at the maximum message size (i.e., quantum = MAX). With these smaller quanta, a *FairnessMeasure* of 0.11 was achieved.

Simulations with presaturated queues were done using the same parameters but without the dDRR scheduler. The results were similar to those depicted in Figure 7 – four flows monopolizes access and all the other flows starved. The *FairnessMeasure*s were close to 4, the worst case for a 4×4 network. Table 1 summarizes the *FairnessMeasures (FM)* and throughputs for the simulations described above. The throughputs are normalized to that achieved when no scheduler is used.

Pictorially, we see that dDRR with smaller quanta distributed access more evenly at small time scales (compare Figures 12 and 13). This is reflected by their *FairnessMeasure*s as shown in Table 1. For short intervals, small quanta yields better *FairnessMeasure* because small quanta enforce finer fairness granularity.

Table 1: Summary of *FairnessMeasure* and throughput performance.

	FM	Normalized Throughput
No scheduler	4.00	1.00
dDRR (quantum = 4 MAX)	0.33	0.86
dDRR (quantum = MAX)	0.11	0.71

We mentioned before that there is an inherent tradeoff between fairness granularity and throughput. This tradeoff is also exhibited in the simulations discussed above. We see that with smaller quanta (hence finer fairness granularity), less throughput was achieved.

To see how fairness granularity is traded off with throughput, another set of simulations was performed. In this set of simulations, all queues were presaturated. All flows are assigned the same quantum. The message lengths are between $MAX/4$ and MAX. The quantum is varied from MAX to 4096 MAX in a power-of-two progression. Figure 14 show how utilization (normalized to the throughput achieved without a scheduler) is traded off with fairness. In Figure 14, we see that as the fairness granularity gets large, we achieve throughput comparable to that achievable without a scheduler.

7 Conclusions

Motivated by our study of the Gemini network architecture, we have presented a distributed solution, dDRR, to a new class of scheduling problems. Namely, that of bufferless, circuit-switched, blocking networks.

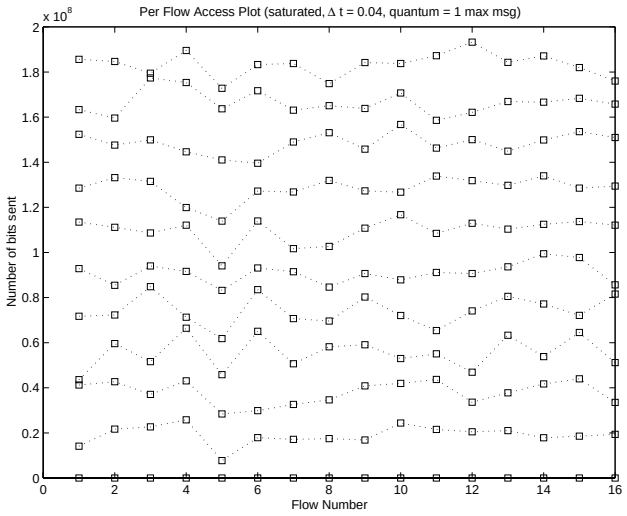

Figure 13: Service amount snapshots at 20 message time interval. Saturated queues, quantum = MAX.

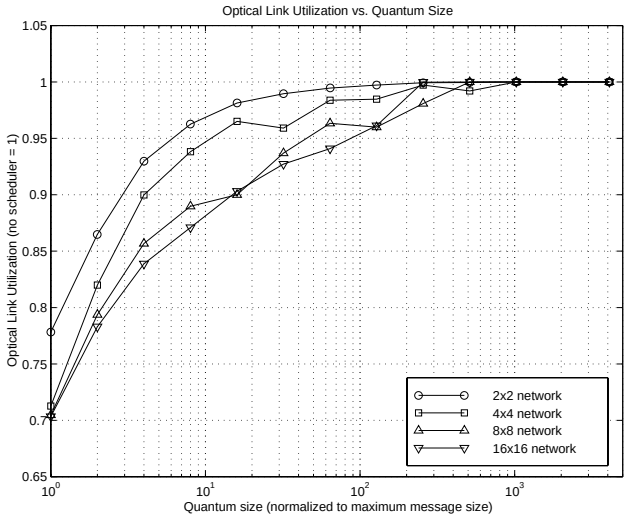

Figure 14: The tradeoff between throughput and fairness granularity of dDRR in Gemini networks.

Assuming variable-length messages, we have shown that there is an inherent tradeoff between throughput and fairness granularity in such networks. We have shown that dDRR has low complexity via analysis and has good fairness measure via simulations. We have also partially characterized the throughput versus fairness granularity tradeoff via simulations.

The dDRR scheduler, as described, can be easily adapted for use in any multistage network topology as long as the path uniqueness property is inherent or enforced. Other results related to head-of-line delay bound, weighted fair scheduling, and the effect of message length distribution on the throughput versus fairness granularity tradeoff are documented in [1].

References

[1] Ch'ng Shi Baw. Design, analysis, and simulation study of optical interconnection networks. Master's thesis, Washington University in St. Louis, 1999.

[2] Ch'ng Shi Baw, R. D. Chamberlain, and M. A. Franklin. The *Gemini* interconnect: Data path measurements and performance analysis. In *Proceedings of Sixth International Conference on Parallel Interconnects*, October 1999.

[3] Ch'ng Shi Baw and M. A. Franklin. An interconnection network simulator. Technical Report WUCCRC 99-03, Washington University in St. Louis, January 1999.

[4] R. D. Chamberlain, M. A. Franklin, R. Krchnavek, and Burak Baysal. Design of an optically-interconnected multicomputer. In *Proceedings of the Fifth International Conference on Massively Parallel Processing Using Optical Interconnections*, pages 114–122, June 1998.

[5] S. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input output queued switch. Technical Report CSL-TR-98-758, Stanford University, 1998.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM '89*, December 1989.

[7] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, June 1994.

[8] Lucent Technologies. Guided Wave Optical Switch Products. Preliminary data sheet, 1997.

[9] Nicholas William McKeown. *Scheduling Algorithms for Input-Queued Cell Switches*. PhD thesis, University of California at Berkeley, 1995.

[10] A. K. Parekh. *A Generalized Processor Sharing Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Massachussetts Institute of Technology, 1992.

[11] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of SIGCOMM '95*, pages 231–243, August 1995.