RRR : Recursive Round Robin Scheduler

Rahul Garg*
rahul@cse.iitd.ernet.in
Tel: +91-11-686-7431, 685-7649
Fax: +91-11-686-8765
Department of Computer Science and Engineering
Indian Institute of Technology
Haus Khas, New Delhi
India

Xiaoqiang Chen
xchen@bell-labs.com
Tel: +1-732-949-2928
Fax: +1-732-834-5379
Bell Laboratories, Lucent Technologies
101 Crawfords Corner Road
Holmdel, NJ 07733
USA

Abstract

Scheduling has been an interesting problem since its inception. In the context of real-time networks, a scheduling algorithm is concerned with dispatching streams of packets sharing the same bandwidth such that certain guaranteed performance for each stream like rate and delay bound is provided. This function has a wide range of applications in network elements such as host adaptors, routers and switches. This paper proposes and describes a new scheduling algorithm named as *recursive round robin* (RRR) scheduler. It is based on the concept of the construction of a scheduling tree in which distinct cell streams are scheduled recursively. Special emphasis is placed on the design and analysis of the scheduler. A delay bound is analytically derived for the scheduler and verified using simulation. The scheduler can work in either a work-conserving mode or non-work-conserving mode. It is shown that the work conserving scheduler is fair. Fairness indexes for the work conserving scheduler are analytically derived. The simple nature of this algorithm makes it possible to implement it at very high speeds, while considerably reducing the implementation cost.

Keywords: Asynchronous Transfer Mode, Scheduling, Quality of Service, Delay Bound, Round Robin, Fairness.

*Corresponding Author

1 Introduction

Scheduling plays an important role in providing QoS guarantees in any data network. The difficulties in the design of a scheduling are largely due to the fact that multiple constraints have to be met, including delay, bandwidth, jitter and loss ratio. In addition, there are a number of desirable properties that a scheduler has to consider. The first property is that a scheduler must isolate one stream from other streams so that it is able to maintain the performance guarantees for such a stream even in the presence of misbehaving streams in the system. The second property calls for a scheduler to divide the available link bandwidth among competing streams in a fair manner. This property is particularly important for the design of work-conserving schedulers. The third property is that a scheduler must perform reasonably well with both, i.e. the number of packet streams and a wide range of link capacities. Finally, a practical scheduler should easily be implemented. In case of scheduling ATM cells, available time for completing a scheduling decision is inversely proportional to the link capacity. This time is only 680 nsec on an OC-12 link, and hardware solution is desirable in this case.

Many of the scheduling algorithm proposed, were first designed for scheduling packets of variable sizes and were complex [4], [13], [11]. Some of them were later adopted for scheduling ATM cells [10]. However the inherent complexity remains and typical operations needed to schedule a cell are addition, multiplication and division. In addition they also need a multi-level priority queue. Among the schedulers optimized for scheduling ATM cells, some have poor delay and fariness properties [5], some are not scalable for fine rate granularity [6] and some may need to over-allocate rate resulting into poor utilization of link bandwidth [3].

In general, schedulers can be characterized as work-conserving or non-work-conserving. A scheduler is said to be work-conserving if the scheduler is never idle when at least one packet is buffered in the system. A non-work-conserving scheduler may remain idle even if there are available packets to transmit.

This paper proposes and describes a new scheduling algorithm called *recursive round robin scheduler* (RRR), which is optimized for scheduling fixed size packets. The proposed scheduler is based on the concept of the construction of a scheduling tree in which distinct streams are scheduled recursively. The rate of each packet stream is represented in binary fraction relative to the link capacity. It can support very large rate granularity for each packet stream. Most of the properties of the scheduler depend upon the count c of number of one in the binary representation of normalized rate allocaetd to the stream. The scheduler provides an upper bound of the delay and delay-jitter for compliant packet streams which are independent of the link utilization. The delay bound is independent of other streams (isolation). The delay bound of a stream depends only on the traffic descriptor and normalized rate allocated to the stream. For a compliant stream of ATM cells of normalized rate r (where r is represented in binary fixed point with g bits), and burst size σ, the scheduler provides a delay and jitter bound of $\frac{1}{r}(\sigma + c)$, where c is the count of number of times one occurs in binary representation of its normalized rate r.

It is shown that work conserving version of the RRR algorithm is fair. The residual link capacity is evenly divided among active streams. The service fairness index of two streams i, j is $(\frac{c_i}{r_i} + \frac{c_j}{r_j})$ where r_i and r_j are (normalized) rates of the two streams, c_i and c_j is the count of number of one in the binary representation r_i and r_j respectively. The worst case fair index of work conserving RRR is bounded by $\frac{c}{r}$.

The algorithm has good scaling properties. Its implementation cost varies linearly with the number of streams and the rate granularity. The algorithm is optimized for fixed packet size and needs only bit manipulation operations to schedule a cell. As a result it can be implemented on high speed low cost ATM host adaptors where the scheduling overhead of the algorithm can be as little as two memory accesses to schedule an ATM cell. The algorithm is also suited for scheduling in high speed ATM switches, especially on the edge of ATM network where traffic policing and isolation is required. Important concepts such as link sharing [1], class based queuing etc. can be possibly be implemented using the RRR scheduler. In fact more complex schedulers can be constructed in which simple and efficient RRR schedulers (alongwith other simple and efficient schedulers) and are used as building blocks so as to meet specific scheduling requirements.

This paper is constructed as follows. In Section 2 description of the RRR scheduling algorithm is presented. Section 3 describes delay and fairness properties of the scheduler. Section 4 discusses variants derived out of RRR. Section 5 describes simulation results. Section 6 discusses implementation issues and tradeoffs for hardware and software implementations of RRR. The paper concludes and describes further work in Section 7. All the proofs are given in section 8.

2 Scheduler Description

We use the term stream to refer to flow or session. The term cell is used to describe the packets of fixed size. Subscript i is usually used to identify a stream. For each stream (i) a rate is allocated at an output link. We refer to this rate as allocated rate (r_i).

We divide all the rates by the link rate. As a result the link rate becomes equal to 1. We refer to the rate so obtained as normalized rate. Further we will assume that the rate is represented in fixed point binary fraction with g bits of granularity. Rate allocated by the scheduler r_i is hence represented by an array of g bits $b_{i1}, b_{i2}, b_{i3}, \ldots, b_{ig}$, and $r_i = \sum_{k=1}^{g} b_i k 2^{-k}$. Another assumption is that sum of rates from input streams is exactly equal to the output link capacity which is 1. In order to achieve this, we add a null stream (or stream 0), with an appropriate rate such that the sum of the rates of all the streams (including the null stream) becomes equal to 1. In the output schedule, whenever the null stream occurs, either the link is left idle or a cell from non real-time class (for example the best-effort traffic) is scheduled, or cell from the "next stream" is scheduled.

2.1 Basic Scheduler

We now describe the scheduler itself, initially by examples and finally by an algorithmic description. The scheduler is constructed in the form of a binary tree (also referred as a scheduling tree). The method of constructing the scheduling tree, for the time being, is deferred to a later stage.

Let's begin with some simple examples to illustrate the basic idea of a RRR scheduler. The trivial case is when the scheduler has no incoming streams. The corresponding tree is shown in Figure 1. The output schedule is simply $0, 0, 0, \ldots$.

On the arrival of a new constant stream of rate 0.5 the root node of the tree *splits* into two nodes as shown in Figure 2. The output schedule in this case is $0, 1, 0, 1, \ldots$.

As a new stream (stream 2) of rate 0.25 arrives, the tree takes the shape as shown in Figure 3. In this case, the output schedule of the RRR would be $0, 1, 2, 1, 0, 1, 2, 1 \ldots$.

A more general case is depicted in Figure 4, where stream i has a rate of 2^{-l_i}. In this case, a leaf of level i of the scheduling tree is allocated to stream i. If no such leaf exists, then a node labeled 0 can be split into two leaves. This process proceeds recursively until a leaf at a desired level is created. The algorithm for constructing the tree (adding stream) is isomorphic to the binary subtraction algorithm. Equally the procedure of deleting stream is isomorphic to the binary addition algorithm.

In the most general case, as shown in Figure 5, a node of the scheduling tree at level l_i is allocated to a stream iff l_ith bit in the stream's rate representation is 1. In other words, if the rate of a stream is equal to $\sum_{k=1}^{g} b_k 2^{-k}$, then a leaf of level k will be allocated to the stream iff b_k is equal to 1.

Thus, given a set of rates, it is possible to construct a tree which satisfies the above described property. The tree has two kind of nodes, internal nodes and leaf nodes. A stream number is associated with each leaf node (in other words, the leaf is allocated to the stream). There is a bit associated with each internal node which keeps track of its subtree visited last. This bit is initialized to 0. In order to schedule a cell, the scheduler starts from the root of the tree and descends down to a leaf, scheduling cells from the stream number associated with the leaf. The scheduler descends down in the left subtree of a node if the corresponding bit is zero. Otherwise it descends down the right subtree. In any case, the scheduler flips the bit associated with each internal node it visits. Thus the first output will be the number stored in the left most leaf and the second output will be the number stored in the left most leaf of the right subtree of the root. Figure 6 gives the algorithmic description of the scheduler.

It should be noted that the service rate of a node m at level l_m is equal to 2^{-l_m}. A node m is visited every 2^{l_m} time units. A cell from the stream associated with leaf m is periodically scheduled in the interval of 2^{l_m} time units.

Two terms, level and phase, associated with each node of the tree are now formally defined.

Definition 2.1 (Level) *Level $l(m)$ of node m is defined as the distance of the node from the root of the tree.*

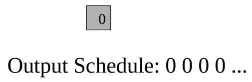

Figure 1: Scheduling tree (no stream)

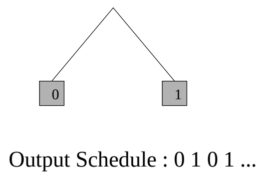

Figure 2: Scheduling tree (1 stream)

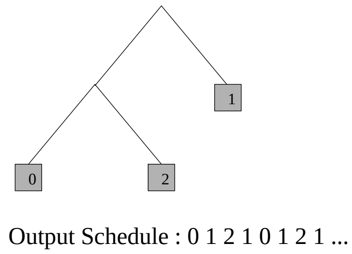

Figure 3: Scheduling tree (2 streams)

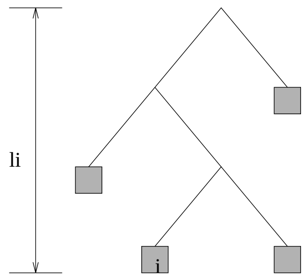

Figure 4: Scheduling tree (general case 1)

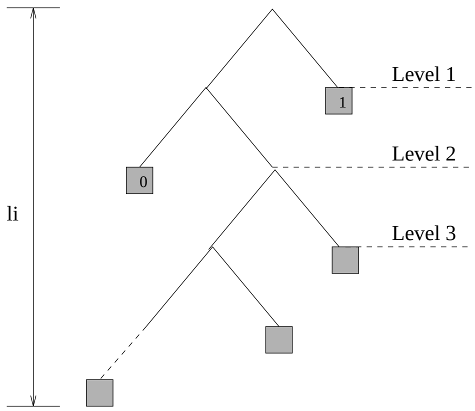

Figure 5: Scheduling tree (general case 2)

The level of the root is zero. The level of a child is one more than that of its parent.

Definition 2.2 (Phase) *Phase $\phi(m)$ of a node m is the slot in which the node is serviced.*

The phase of the root node is 0. The phase of the left child (m) of a node (n) is equal to that of the parent node. Mathematically, if node m is a left child of node n then, $\phi(m) = \phi(n)$. The phase of the right child (m) of node (n) is $2^{l(n)}$ more than that of the node. Formally, if node m is the right child of node n then $\phi(m) = \phi(n) + 2^{l(n)}$.

A cell from the stream associated with a leaf m is scheduled in slots $\phi(m), \phi(m) + 2^{l(m)}, \phi(m) + 2 * 2^{l(m)}, \phi(m) + 3 * 2^{l(m)}, \ldots$[1].

2.2 Adding and Deleting Streams

As mentioned earlier, adding a stream with a given allocated rate is isomorphic to the deletion algorithm for binary numbers. One can view the addition of a stream with rate r as taking away r units of rate from the null stream. Similarly deleting a stream of rate r is isomorphic to binary addition, as r units of rate become free and get added to the rate of the null stream.

The scheduling tree is modified dynamically as streams are added or removed. However any modifications to the scheduling tree must be made carefully so that the delay bounds of the existing streams are not violated. To achieve this, we need to define the following for later use.

Definition 2.3 (Unfragmented rate allocation) *A stream i is said to have unfragmented rate allocation iff for any level l of the scheduling tree, there is at most one node allocated to the stream i.*

The basic idea is to ensure all the streams (including the null stream) have unfragmented rate allocation each time a stream added or deleted from the scheduling tree. Unfragmented rate allocation bounds the number of leaves allocated to a particular stream. If the height of the scheduling tree is k and a stream has unfragmented rate allocation, then at most k nodes in the scheduling tree can be allocated to the stream. We will show that the number of leaves is directly related to delay and fairness properties. It also limits the size of the scheduling tree, and hence the amount of state and complexity of implementation.

To add a stream i with the requested rate r_i, the first step is to check if the requested rate is available. If the link doesn't have adequate rate, the requested stream is denied to be added to the scheduling tree. If the link has adequate rate, i.e., $r_i \leq r_0$, then we proceed further.

Assume

$$r_i = \sum_{k=1}^{g} b_{ik} 2^{-k}$$

The procedure begins with the least significant bit ($k = g$ *downto* 1) of the requested rate, from a node at the largest level. If b_{ik} is zero, no actions need to taken but k is simply decremented. If both b_{ik} b_{0k} are equal to 1, then the node at level k previously allocated to the null stream is allocated to the current stream i. This requires simple relabelling in the scheduling tree.

If $b_{ik} = 1$ and $b_{0k} = 0$, however, a *borrow* operation is performed from level k of the scheduling tree. During this operation, we move up to level $k-1$ in the scheduling tree and find if there is a leaf allocated to the null stream at $k-1$ level. If the answer is no, we move up the scheduling tree again till such a leaf is found. It is always possible to find such a leaf if $r_i \leq r_0$ and the null stream has unfragmented rate allocation. After a leaf allocated to the null stream at level k' is found, it is *split* into two children at level $k'+1$. One of the children is allocated to the null stream while the other child is used for performing the *split* operation again. This process is continued till level k is reached. At level k, one of the two new children is allocated to stream i and the other to the null stream.

The above procedure is continued till the root of the scheduling tree is reached. r_0 is decremented by r_i to update the residual bandwidth ($r_0 = r_0 - r_i$). It should be noted that at this stage, the scheduling tree has unfragmented rate allocation for all the streams.

The above procedure of adding streams is illustrated by examples in Figure 7, Figure 8 and Figure 9. In these examples, it is assumed that 4 bits are used to represent a rate.

Evolution of the scheduling tree when stream 1 of rate 0.0101 is added for scheduling on an idle link is shown in Figure 7(a) to (f). Initially there is no stream, so all the bandwidth is allocated to the null stream. The scheduling tree is single leaf as shown in (a). Since 4th bit of rate of stream 1 is 1, a node at level 4 must be allocated. Since there is no leaf allocated to the null stream at level 4, the borrow operation is performed. The first leaf allocated to the null stream is the root itself. Thus the root is split into two children at level 1. One of the children is allocated to the null stream and the other is marked with B representing the borrow operation, as shown in (b). The node marked B is split again as shown in (c). This borrow operation is carried as shown in (d) and (e). As the leaf node in (e) is split again,

[1] The slots mentioned here are not the only slots when cells from the stream are scheduled. If more than one node of scheduling tree are allocated for a stream, then there are additional slots where cells from the same stream are scheduled

two leaves at level 4 associated with the null stream are created. One of the leaf is allocated to stream 1. Now there is a leaf at level 2 allocated to the null stream. This leaf is allocated to stream 1 resulting in the scheduling tree as shown in (f).

Figure 8 and Figure 9 are self explanatory as the borrow operation is not needed in allocating the nodes, while adding stream 2 of rate 0.0100 and stream 3 of rate 0.0011.

To delete stream i, all its allocated nodes are reallocated to the null stream. At this stage, the null stream may have fragmented rate allocation. In order to unfragment the rate allocation of the null stream, the inverse of the split operation (*join*) and of the borrow operation (*carry*) has to be performed.

If there are two nodes at the same level assigned to the null stream, they should be joined together to form a parent at a higher level. The join operation can be applied only on siblings (two nodes with a common parent). Assume that the n_1 and n_2 are the nodes allocated to the null stream at the same level say l. If n_1 and n_2 are siblings then the join operation is directly applied and the two nodes are replaced by a leaf node at a lower level. In case n_1 and n_2 are not siblings. Let n_3 be the sibling of n_1. Now the nodes n_2 and n_3 are swapped. This operation is called *phase exchange*[2]. After the phase exchange, n_1 and n_2 become siblings and the join and carry operation is performed to unfragment rate allocation of the null stream.

The above procedure is carried out from the highest level to the root of the scheduling tree, until the rate allocation of the null stream becomes unfragmented. Finally r_0 is updated to the value of the currently available bandwidth ($r_0 = r_0 + r_i$).

In continuation with the examples described in Figure 7 to Figure 9, we illustrate deleting stream 1 in Figure 10. The initial scheduling tree is shown in (a). In order to delete stream 1 all the leaves allocated to it are allocated to the null stream as shown in (b). Now there are two leaves at level 2 allocated to the null stream. Thus the null stream is now fragmented. In order to unfragment the null stream, the phase exchange between two leaves (one allocated to stream 2 and the other to stream 0) at level 2 is carried out, resulting in the scheduling tree as shown in (c). Finally the two siblings allocated to the null stream are merged together to move up, resulting in a leaf allocated to the null stream at level 1 as shown in (d).

[2]The phase exchange has to be carried out carefully such that it doesn't affect the delay bound and other properties of the stream associated with n_2. The phase exchange is carried out in two steps. Firstly node n_3 allocated to null stream is reallocated to the stream associated with n_2. At this stage the stream gets more share of link rate. If current time is t then node n_2 is reallocated to null stream at time $t + 2^l$. This deallocation can be done asynchronously as the second step.

```
Scheduler () {
        for_each(node) /* Initialization */
                node.bit = 0;
        for_ever
                schedule_one_cell (root);
}

schedule_one_cell (node) {
        if (node == leaf)
                output (node.stream_number);
        else_if (node.bit == 0) {
                node.bit = 1;
                schedule_one_cell (node.left_child);
        }
        else_if (node.bit == 1) {
                node.bit = 0;
                schedule_one_cell (node.right_child);
        }
}
```

Figure 6: The RRR scheduling algorithm

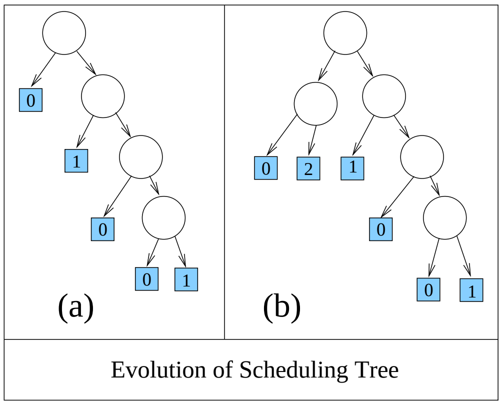

Figure 8: Adding stream of 2 rate 0.0100

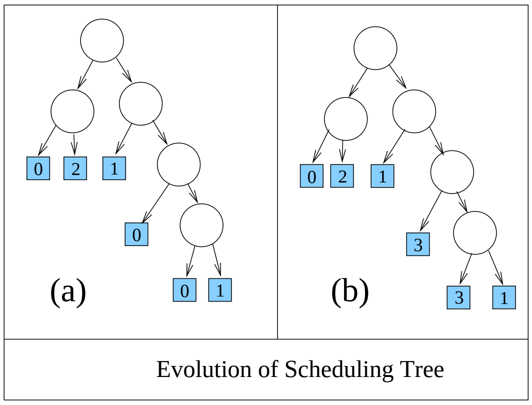

Figure 9: Adding stream 3 of rate 0.0011

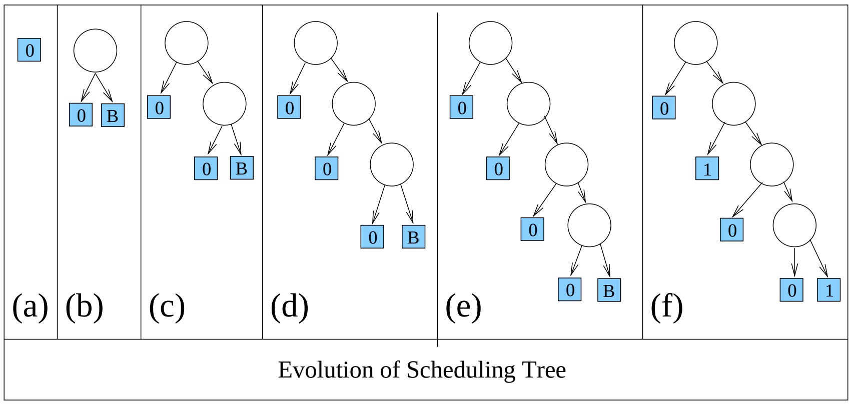

Figure 7: Adding stream 1 of rate 0.0101

Now all the streams have unfragmented rate allocation.

2.3 Allocation for Floating Point Rates

In this section we will discuss the rate allocation for the scheduling tree when the rate is represented in the form of floating point. Let the rate $r \leq 1$ be represented in simple binary floating point with g bits of mantissa and e bits of unsigned exponent. Let E be the value of exponent. Now $r = 2^{-E} \sum_{k=1}^{g} \frac{b_k}{2^k}$. The allocation for this rate is possible if the height of the scheduling tree is $E + c$. If $E_{\max}$ is the maximum value which the exponent can take, then the rate allocation as discussed earlier is possible provided the height of the scheduling tree is $E_{\max} + c$.

It is important to note that even if the height of the scheduling tree has been increased to a larger value, the number of nodes allocated for a stream is still bounded by g, the mantissa (granularity) of rate representation. Keeping this fact in mind is possible to implement the scheduler in hardware as well as software with reasonable complexity. The implementation of the floating point rate representation will be very similar to that of the basic scheduler. The bounds and other properties of the scheduler change marginally if the floating point rate allocation is carried out.

In the rest of the paper, the fixed point rate representation is used to simplify descriptions and proofs.

2.4 Work Conserving RRR

We refer to the scheduler discussed before as *basic scheduler* or *basic RRR.* The obvious extension of the basic scheduler is to a work conserving scheduler.

In the basic RRR scheduling, the time slots which a stream gets is fixed once the allocation of leaves for that stream has been completed. The slots are inde-

```
Scheduler () {
        for_each(node) /* Initialization */
                node.bit = 0;
        for_ever
                schedule_one_cell (root);
}

schedule_one_cell (node) {
        if (node == leaf) {
                if (there are no pending cells)
                  return; /* Scheduler is idle */
                if there is a pending cell on
                         stream (node.stream_number)
                  output (node.stream_number);
                else
                  schedule_one_cell (root)
        else_if (node.bit == 0) {
                node.bit = 1;
                schedule_one_cell (node.left_child);
        }
        else_if (node.bit == 1) {
                node.bit = 0;
                schedule_one_cell (node.right_child);
        }
}
```

Figure 11: Work Conserving RRR

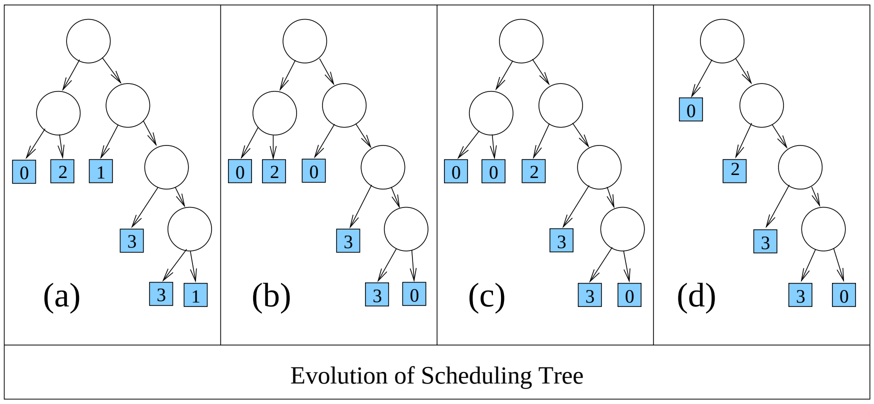

Figure 10: Deleting stream 1

pendent of traffic on other streams. It might happen that a scheduling slot of a stream arrived but there is no pending cell to be scheduled. In such a case no cell will be scheduled if the RRR is used. The work conserving RRR will however scheduler a cell from the next non back-logged stream. Figure 11 describes the work conserving variant of RRR.

2.5 Discussion

In this section the main idea behind the scheduler has been presented. The link bandwidth is partitioned into two equal halves by splitting the node at root of the scheduling tree into two nodes at level 1. While scheduling, these two nodes are visited in round robin order. Both the nodes at level 1 can be further split into two nodes each at next level, further partitioning the link bandwidth. Continuing in this manner, the link bandwidth can thus be partitioned hierarchically giving rise to a scheduling tree. Traversal of nodes of the tree becomes *recursive round robin.*

The concept of scheduling tree is abstract. It is not always necessary to construct the scheduling tree for generating the schedule. Hardware implementation sketched in section 6 doesn't explicitly store the scheduling tree data structure. Therefore one must not judge the time complexity of the scheduler by the time complexity of algorithm previously described in Figure 6. Several implementations of the algorithm are possible and the tradeoff is discussed in section 6. We believe that efficient hardware implementation of the scheduler can schedule one cell per clock cycle. The amount of hardware needed scales linearly with the rate granularity g.

3 Main Properties of RRR Scheduler

It will be shown shortly that all the properties of RRR scheduler scale linearly with granularity of rate allocation, g. To be more percise, for a stream i all the properties depend on c_i, which is count of number of ones in binary representation of r_i. c_i is bounded by granularity of rate allocation, g. For simplicity we will only discuss the g-bit fixed point rate representation of a stream scheduled by the RRR scheduler. The floating point rate representation however can be dealt with in a similar manner. We first evaluate bound for a single node in the network and later extend it for multiple nodes.

The notations used are summarized in Table 1. Let $S(t)$ be equal to the number of cell of the stream scheduled by the basic RRR scheduler in the interval $[0, t]$. Then

$$S_i(t) \quad \leq \quad \sum_{k=1}^{g}[\lfloor\frac{t-\phi(b_{ik})}{2^k}\rfloor + 1]b_{ik} \tag{1}$$

where b_{ik} is k^{th} bit in the normalized rate (r_i) allocated to stream i, and $\phi(b_{ik})$ is phase of corresponding the k^{th} bit. The right hand side of equation 1 is involved in all the proofs. Intutively, the contribution to $S_i(t)$ is only from terms in the summation where b_{ik} is non zero. For each k such that b_{ik} is equal to 1, cells are scheduled at time instants $\phi(b_{ik}), \phi(b_{ik}) + 2^k, \phi(b_{ik}) + 2 * 2^k, \ldots$. Summing this up gives the equation 1.

3.1 Delay and Buffer Bound

For a stream, say i which is continuously backlogged in the interval $[0, t]$ we have

$$S_i(t) \quad \geq \quad \sum_{k=1}^{g}[\lfloor\frac{t-\phi(b_{ik})}{2^k}\rfloor + 1]b_{ik} \tag{2}$$

Symbol	Meaning
$A_i(t)$	Number of cells of stream i arriving in interval $[0,t]$
$S_i^j(t)$	Number of cells of stream i scheduled at node j in time $[0,t]$
$B_j(t)$	Backlog of stream i at time instant t
$b_j(t)$	Burstiness function of stream i
r_i^j	Normalized rate allocated to stream i at link j
c_i^j	Count of number of 1s in r_i^j.

Table 1: Notations

Equation 2 is true for both, basic RRR scheduler and work conserving RRR. The equation is true even in the presence of phase exchanges described in section 2.2. This gives the following upper bound (d_i) for delay:

$$d_i \quad \leq \quad \frac{1}{r_i}(\sigma_i\max + c_i) \tag{3}$$

where c_i is the count of number of times 1 occurs in the binary representation of normalized rate r_i and $\sigma_i\max$ is the burst size in the leaky discripor of the stream. Mathematically $\sigma_i\max = \max_t[b_i(t) - r_i t]$. The proof is given in section 8.1.

The above delay bound is close to that of of "optimal fair schedulers" like WFQ if the burstiness $\sigma_i\max$ is very large as compared to the paremeter c_i. The delay bound for WFQ is $\frac{1}{r_i}(\sigma_i\max + 1)$. For instance assume a stream of MPEG video of rate 5 Mbps ($r = 11.8K$ ATM cells per second), and burst size of $10KB$ (189 cells) is scheduled by RRR scheduler with 16 bits of rate allocation. The total delay bound would be $17.37ms$. The corresponding bound for WFQ scheduler is $16.10ms$.

The delay bound for a network of basic RRR schedulers is given by the following equation

$$d \leq \frac{1}{r}(\sigma\max + \sum_{k=1}^{N} c^k) \tag{4}$$

assuming that the stream goes through a network of N nodes, r^j is the normalized rate allocation of stream at node j and c^j is the corresponding count of number of ones. The proof is in section 8.2.

The output stream of basic RRR scheduler is smooth. As a result a bound on number of buffers needed at an intermediate node to guarantee zero cell loss is obtained. At link j, $c^{j-1} + c^j$ cells of buffer is needed to guarantee zero cell loss. Proof of the above bound is given in section 8.3. Thus, if 20 bits of rate representation is used, then an average of 20 cells of buffer per stream would be needed at intermediate nodes.

3.2 Fairness Properties

In case a link is under-utilized by streams scheduled on it, the remaining link capacity should be distributed to active (or backlogged) streams fairly, i.e. in proportion of their allocated rate [7]. Ideally, if stream i and stream j are continuously backlogged in the interval $[t_1, t_2]$ then amount of service in the interval should be related by:

$$\frac{S_i(t_1,t_2)}{r_i} = \frac{S_j(t_1,t_2)}{r_j} \tag{5}$$

In practice it is impossible to achieve ideal fairness because of cell boundaries. So a measure of fairness, called "service fairness index" (SFI) is defined to quantify the fairness.

$$SFI = \max_{t_1,t_2} \mid \frac{S_i(t_1,t_2)}{r_i} - \frac{S_j(t_1,t_2)}{r_j} \mid \tag{6}$$

Small values of SFI correspond to more fairness. $SFI = 0$ corresponds to ideal fairness.

The SFI for work conserving RRR is given by:

$$\frac{c_i}{r_i} + \frac{c_j}{r_j} \tag{7}$$

The proof is given in section 8.4. The bound on SFI increases linearly with increasing granularity, as c_i and c_j are bounded by g.

Another measure of fairness is based on the worst case delay for clearing the backlog of a stream's queue. A scheduler has worst case fair index (WFI) of $\frac{C_i}{r_i}$ for stream i if C_i is the smallest number satisfying the equation:

$$S_i(t_1,t_2) \geq (t_2 - t_1)r_i - C_i \tag{8}$$

The normalized worst case fair index would be C_i. This measure is important in hierarchical scheduling [12]. The delay bound of hierarchical scheduler increases with increasing WFI. The WFI for work conserving RRR scheduler is bounded by $\frac{c_i}{r_i}$. The proof is given in section 8.4.

4 Variants of RRR

Now we are ready to introduce and discuss variants of RRR.

4.1 k-ary RRR

In k-ary RRR scheduler, the scheduling tree is not binary. The degree of the scheduling tree is k. In order to schedule one cell, the tree is traversed recursively starting from the root node. Every non-leaf node contains an index which takes a value from 0 to $k-1$. The index is initialized to 0. For any internal node, the child pointed by the index of the node is recursively visited. The index of the node is incremented by 1 (mod k), each time the node is visited.

4.2 Generalized RRR

In generalized RRR, the degree of nodes at different levels of the tree can be different. For each level l there is a degree parameter d_l. Each internal node at level l has d_l children. The scheduling tree is traversed recursively as in the k-ary RRR scheduler, except that the parameter index of a node at level l can take a value from 0 to $d_l - 1$. The index at level l is incremented modulo d_l.

4.3 Recursive Tree Based Scheduling

This is the most general form of recursive scheduling. In this case the scheduling tree can be arbitrary. Each node in the tree contains the count of the number of its children. In order to schedule a cell the tree is traversed recursively. While visiting an internal node, its index is incremented modulo its number of children. The tree need not be static. New internal nodes of different degrees can be created when needed. The nodes at various parts in the tree can be brought together and combined into a single node.

4.4 Hierarchical Scheduling

The schedulers discussed so far schedule streams on an output link. It may be useful to schedule a set of input streams to get an output stream. There may be a number of such schedulers generating a number of output streams. These streams can in turn be scheduled by another scheduler on a link. This process can be repeated hierarchically over a number of levels. The motivation behind building such schedulers is to use simple schedulers like static priority, RRR etc. as basic building blocks so as to construct hierarchical schedulers which meet the desired requirements. This idea first appeared in *rate control static priority* scheduler as described in [11] and was later refined and formalized for the fair queuing algorithms in [12]. Work conserving RRR has a bounded worst case fairness index (WFI). Therefore, the corresponding hierarchical scheduler approximates H-GPS. As a result, RRR can be directly applied to implement concepts such has link sharing.

Finally it is noted that the variants of RRR discussed above in this section share most of the good properties of the basic RRR algorithm outlined in the paper, namely, simplicity, support for wide rate range, ease of hardware and software implementation and possibility of distributed implementation. The proofs of fairness and delay bound can be easily extended to these variants.

5 Simulation Results

The simulation scenario is depicted in Figure 12. Input streams are chosen from a uniform distribution between 1 and 6Mbps (6Mbps is expected bit rate of MPEG-II streams). The streams are scheduled on an output link of 156Mbps. Fifteen bits were used to represent a rate. The fifteen bits was chosen to ease the simulation on 32 bit computers. Further, the purpose of simulation was to experimentally verify the correctness of the proof. Note that the basic RRR algorithm provides perfect isolation. The delay of a cell of a stream is independent of arrival pattern of cells of other streams. Therefore the delay distribution of cells of a single stream will remain same irrespective of the cross traffic.

While constructing the scheduling tree, the phase at each level was assigned randomly. The experiment was repeated a number of times to make sure that the results are consistent and the simulations are correct. Figure 13 shows the delay of successive cells of a stream for a part of a simulated period.

Figure 14 plots the statistical delay distribution of cells of the stream. The X-axis represents the delay of cells which varies from zero to the theoretical delay bound. The Y-axis plots the cumulative number of cells having delay in the corresponding range. It was found that with the random phase allocation, the delay bound was never reached. However with a specific phase allocation the tail of the delay distribution approached close to the theoretical delay bound.

6 Implementation Considerations

It should be noted that the number of leaves in the tree is bounded by g times the number of streams being scheduled (N). Therefore the space requirement for the scheduler to store the scheduling tree is bounded by $2Ng$.

In order to schedule a cell, one must descend down the scheduling tree to a leaf. The height (the maximum level of a node in the tree) of the tree is bounded by g. Thus, in the worst case, a naive implementation will take no more than g operations to schedule a single cell.

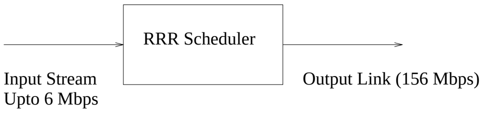

Figure 12: Simulation scenario

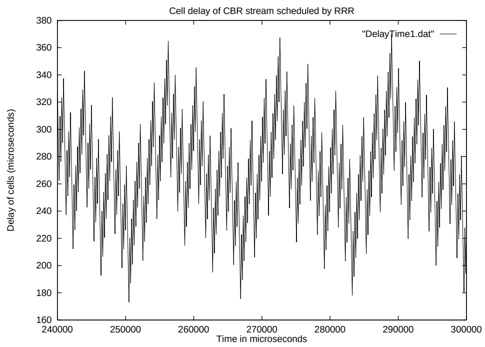

Figure 13: Delay of successive cells

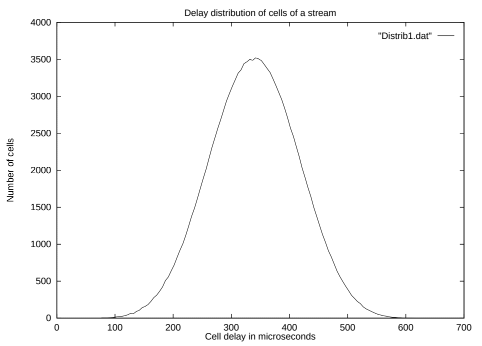

Figure 14: Distribution of delay of cells

However, clever implementations can give better performance in terms of the time needed to schedule a cell. Given a binary RRR scheduling tree with any rate allocation for streams, it is straightforward to construct a corresponding 4-ary RRR scheduling tree such that the schedule generated by the binary and 4-ary schedulers on the corresponding tree is identical. The tree is constructed starting from the root node and collapsing two levels of the binary RRR tree into a single level of 4-ary tree. In case that there is a leaf node in the first level of the binary RRR tree, two copies of the node are made in the 4-ary tree. The four children of the root node in the 4-ary scheduling tree are positioned so that they are visited in the same order as in the binary RRR tree. This process is continued till all the nodes (including leaves) of the binary RRR tree are collapsed into the 4-ary scheduling tree.

The height of 4-ary tree will be $\frac{g}{2}$. As a result, $\frac{g}{2}$ operations are needed in order to schedule one cell. This improvement in the number of operations is at the cost of additional space requirements. The number of leaves in 4-ary scheduling tree are bounded by two times the number of leaves of corresponding binary tree i.e. $2Ng$. The number of internal nodes in the 4-ary tree will be bounded by $\frac{4}{3}2Ng$. There is a $\frac{4}{3}$ factor increase in storing space to reduce the number of operations in scheduling a cell by factor of two. For any software implementation, there is a tradeoff between time required to schedule a cell, amount of memory available to scheduler to maintain its data structures.

For the fixed point rate representation, both the space requirements and the number of operations to schedule a cell are dependent on the maximum number of streams to be scheduled (N). And the the rate granularity (g) is clear from the above expressions. Similarly for the floating point rate representation, the rate range (the maximum value the exponent of rate can take) will also be a factor in the dependence.

There are similar tradeoffs and dependencies in hardware implementation as well. Consider a counter based hardware implementation of the RRR scheduler. For simplicity, we assume the fixed point rate representation. For each leaf node in a binary scheduling tree, there will be a counter and a register, which stores its level, phase and associated stream number. All the registers output their stream numbers which are fed to a hardware multiplexer. All the counters are clocked by the same clock and are incremented every clock cycle. When a counter reaches the phase stored in the corresponding register, the multiplexer is activated to select the stream number of its corresponding register. Every counter is reset to zero when its value becomes 2^{level}.

Since there are Ng leaves in a scheduling tree, there will be Ng counters and the same number of associated registers. Each counter should be able to count up to $2^g - 1$. Each register will store the phase information of g bits, the level information of $\log g$ bits and the stream number of $\log n$ bits. One $\log N$ bit multiplexer is needed which should be able to multiplex and output a stream number from Ng registers.

The amount of logic needed for this implementation is $Ng(g + \log g)$ for the counters, $Ng(g + \log N)$ for the registers and $2Ng \log N$ for the multiplexer. The speed of this hardware implementation depends upon the multiplexing delay and the clock skew. The multiplexing delay as well as the clock skew depends upon Ng. There are Ng distinct counters running by the same clock, which causes the clock skew. There are Ng outputs of registers from which the multiplexer chooses one. Hence the multiplexing delay depends heavily on Ng.

7 Conclusion and Future Work

A simple and new scheduling algorithm called recursive round robin (RRR) has been described in this paper. This scheduler can be used to schedule streams on a link with certain constraints. The scheduling algorithm guarantees a delay and jitter bound on compliant streams. Proofs of delay bound and jitter bound are provided for generic traffic descriptors. Several variants of the RRR scheduling algorithm are described. It has been shown that the work conserving version of the RRR is fair. Bounds on two different fairness indexes are analytically derived. It would be easy to generalize this result to other work conserving variants of RRR. Proof of worst case delay bound remains valid for the work conserving RRR. The procedure of dynamic addition and deletion of streams with the floating point rate representation has also been presented. Implementation tradeoffs for hardware and software implementation have been discussed.

Proof of delay bound has been supplemented by simulations. The delay bound proven for the RRR scheduling algorithm is tight but there is a room for further improvement by finding *efficient node allocation strategies*. Another area that this work can further be extended is to study the detailed performance for those proposed variants of the basic RRR.

Further work is also needed to extend the basic RRR such that it will be able to schedule packets of variable length.

Because of simple nature and provable bounds the proposed algorithm in this paper may find applications at a number of places. Since the algorithm is simple and neat, it is possible to implement it in a distributed environment. This feature widens the scope of implementations based on this algorithm.

The scope of implementations based on this algorithm include (but necessarily not limited to) scheduling of packets on networks which provide guaranteed performance. Some of the examples are microprocessor based or hardware based implementations on ATM/TCP-IP Network Interface Cards (NIC), ATM Switches, IP Routers.

The applications of distributed implementations based on the RRR scheduling algorithm are in building real-time networks which provide QoS guarantees on a shared physical media. Broadcast and multicast can also be supported if the RRR is used. The shared communication medium may have asymmetric properties (e.g. cable modems) in uplink and downlink paths. An implementation of RRR on traditional Ethernet LANs can build real-time LANs which would provide QoS guarantees. Other examples of shared medium are Ethernet (Wireless or wired), Token Ring, Wireless ATM etc. This work has a potential to make a significant impact in wireless communication technologies like cellular telephones, cellular data, wireless in local loop, digital TV broadcast, wireless ATM.

8 Appendix

We use the notation shown in Table 1.

8.1 Delay Bound for Single Node

Let $b(t)$ be the bounding burstiness function of the stream under consideration (see [2]). Let r be the allocated rate of RRR scheduler for scheduling the compliant stream[3].

Without loss of generality, assume that a backlog period of the stream starts at time $t = 0$. Let t be any instant in this backlog period. Since the stream is continuously backlogged in the interval $[0, t]$, from Eq. 2 we have

$$S(t) \geq \sum_{k=1}^{g} [\lfloor \frac{t - \phi(b_k)}{2^k} \rfloor + 1] b_k \tag{9}$$

$$\geq \sum_{k=1}^{g} [\frac{t - \phi(b_k)}{2^k}] b_k \tag{10}$$

Since the maximum value of phase ($\phi(b_k)$) of a node

[3]The rate r should be chosen carefully while performing admission control as it might affect the network utilization. Some guidelines to choose r are described in [2]. For constant rate streams r could be chosen which is close of r. The rate r of the compliant stream can take arbitrary value in $[0, 1)$. Given rate r, it is desirable to pick $r_a > r$ such that the number of times 1 occurs in the g bit binary representation of r is minimized without adding significantly to the overhead. Such a selection can further minimize the jitter introduced in the stream by the scheduler

is less than its period (2^k), we have

$$\begin{aligned} S(t) &\geq \sum_{k=1}^{g}[\frac{t}{2^k}]b_k - c & (11) \\ &= tr - c & (12) \end{aligned}$$

Let $A(t)$ be the number of cells of a compliant stream arrived in the interval $[0, t]$. Assume that the cell arrived at time t' suffers the maximum scheduling delay in this backlog period. Let d be the delay of cell arriving at time t'. This implies that all the traffic arriving in the interval $[0, t']$ is serviced exactly by time $t' + d$. Therefore,

$$A(t') = S(t' + d) \quad (13)$$

Substituting $S(t)$ from Eq. 12 we have

$$A(t') \geq r(t' + d) - c \quad (14)$$

Since $A(t') \leq b(t')$ we get

$$rd \leq b(t') - rt' + c \quad (15)$$

The burst size is defined as $\sigma_{\max} = \max_t(b(t) - rt)$. Substituting $\sigma_{\max}$ in above equation we get

$$\begin{aligned} rd &\leq \sigma_{\max} + c & (16) \\ \Rightarrow \quad d &\leq \frac{1}{r}[\sigma_{\max} + c] & (17) \end{aligned}$$

Since change of origin only affects the values of phases ($\phi(b_k)$), it can be shifted to the beginning of any backlog period without changing anything in the above proof. After de-normalization, when the unit of rate is cells per second and the unit of $\sigma_{\max}$ is number of cells, the above equation gives the delay bound in seconds.

The worst case delay (for compliant streams) is given by Eq.(17). It must be noted that in Eq.(2) used in proving the delay bound is a lower bound on $S(t)$. Therefore the same proof applies to the work conserving RRR and the RRR with dynamic addition and removal of streams (This inequality would hold true in *phase shift* operation needed to unfragment the rate of the null stream).

8.2 Delay Bound for Multiple Nodes

We prove the end-to-end delay bound for a network of RRR schedulers using the theory of latency rate servers [9]. Let a backlog period of a stream start at time t. Let t' be another time instant in the same backlog period. According to the results presented in [8], if a server guarantees that the amount of stream traffic served in the interval $[t, t']$ is at least $r(t' - t - \theta)$ then the server is a latency rate server of latency θ and rate r. It was also shown in [8] that the end-to-end delay bound for a leaky bucket compliant stream of parameters (σ, r), serviced by a network of N latency rate servers is $\sigma/r_a + \theta_s$, where r_a is the minimum rate of the N servers ($r_a \geq r$) and θ_s is the sum of the latencies of the N latency rate servers.

In order to prove the end-to-end delay bound, we only need to show that the RRR scheduler is a latency rate server. Equation 12 may be rewritten as

$$S(t) \geq r(t - \frac{c}{r}) \quad (18)$$

where t is any time instant in the backlog period of the stream starting at time 0. Since the origin may be shifted to the beginning of any backlog period, we conclude that the RRR scheduler is a latency rate server with rate r and latency c/r. Therefore the end-to-end delay bound of a stream served by a network of N RRR schedulers is given by

$$d \leq \frac{1}{r_a}(\sigma_{\max} + \sum_{k=1}^{N} c^k) \quad (19)$$

where r_a is the minimum of the rates allocated to the stream at N nodes, and c^k is the number of times one appears in the normalized rate allocated to the stream at the kth node. Again note that this bound is valid for basic RRR as well as work conserving RRR.

8.3 Buffers Needed at Intermediate Nodes

Using the results from the theory of latency rate servers [8] , the maximum amount of buffers needed at kth node is $\sigma_{\max} + \sum_{i=1}^{k} c^i$.

However, if the non work conserving RRR is used at every node, and the rate (r^k) allocated at a node (k) is no less than the rate allocated (r^{k-1}) at its upstream node ($k-1$) (i.e. $r^k \geq r^{k-1}$), then better buffer bounds can be obtained.

At node k the arrival $A^k(t)$ satisfies

$$A^k(t) \leq r^{k-1}t + c^{k-1}$$

The output $S^k(t)$ satisfies

$$S^k(t) \geq r^k t - c^k$$

Therefore maximum backlog $B^k(t)$ is bounded by $S^k(t) - A^k(t)$. Since r^k is greater than or equal to r^{k-1},

$$B^k(t) \leq c^{k-1} + c^k$$

$$B(t) \leq 2g$$

at any intermediate node in the network.

8.4 Fairness of Work Conserving RRR

Now we show that the work conserving RRR is fair and we derive bounds for two fairness measures of the scheduler. If two streams i and j are continuously backlogged in the interval $(t_1, t_2]$ and the rate allocated to them is r_i and r_j and $S_i(t_1, t_2)$ and $S_j(t_1, t_2)$ is the number of cells of stream i and stream j scheduled by the RRR scheduler in the interval $(t_1, t_2]$ then SFI is defined as:

$$\mathrm{SFI} = \max_{t_1, t_2} \mid \frac{S_i(t_1, t_2)}{r_i} - \frac{S_j(t_1, t_2)}{r_j} \mid$$

Without loss of generality assume that SFI is maximized when $t_1 = 0$. Let $S_i(t)$ stand for $S_i(0, t)$. Let $v(t)$ be the number of leaves of the scheduling tree visited in time $(0, t]$. $v(t)$ is similar to the notion of virtual time used in the context of fair schedulers (like weighted fair queueing). Note that $v(t) = t$ if all the streams are continuously backlogged in the interval $[0, t]$. Now

$$\begin{aligned}
S_i(t) &= \sum_{k=1}^{g} [\lfloor \frac{v(t) - \phi(b_{ik})}{2^k} \rfloor + 1] b_{ik} && (20)\\
&\leq \sum_{k=1}^{g} [\frac{v(t) - \phi(b_{ik})}{2^k} + 1] b_{ik} && (21)\\
&\leq v(t) [\sum_{k=1}^{g} \frac{b_{ik}}{2^k}] + g && (22)\\
&= v(t) r_i + g && (23)\\
\Rightarrow \frac{S_i(t)}{r_i} &\leq v(t) + \frac{g}{r_i} && (24)\\
S_j(t) &= \sum_{k=1}^{g} [\lfloor \frac{v(t) - \phi(b_{jk})}{2^k} \rfloor + 1] b_{jk} && (25)\\
&\geq \sum_{k=1}^{g} [\frac{v(t) - \phi(b_{jk})}{2^k}] b_{jk} && (26)\\
&\geq v(t) [\sum_{k=1}^{g} \frac{b_{jk}}{2^k}] - g && (27)\\
&\geq v(t) r_j - g && (28)\\
\Rightarrow \frac{S_j(t)}{r_i} &\geq v(t) - \frac{g}{r_i} && (29)\\
\Rightarrow \frac{S_i(t)}{r_i} - \frac{S_j(t)}{r_j} &\leq g(\frac{1}{r_i} + \frac{1}{r_j}) && (30)\\
\Rightarrow \mathrm{SFI} &= g(\frac{1}{r_i} + \frac{1}{r_j}) && (31)
\end{aligned}$$

Equation 31 gives a bound on service fairness index of work conserving RRR.

We now derive a bound for worst case fair index (WFI) of the scheduler. If C_i is the smallest number satisfying the equation:

$$S_i(t_1, t_2) \geq (t_2 - t_1) r_i - C_i \qquad (32)$$

then WFI $= C_i$.

$$\begin{aligned}
S_i(t) &= \sum_{k=1}^{g} [\lfloor \frac{v(t) - \phi(b_{ik})}{2^k} \rfloor + 1] b_{ik} && (33)\\
&\geq \sum_{k=1}^{g} [\frac{v(t) - \phi(b_{ik})}{2^k}] b_{ik} && (34)\\
&\geq t [\sum_{k=1}^{g} \frac{b_{ik}}{2^k} b_{ik}] - g && (35)\\
&\geq t r_i - g && (36)
\end{aligned}$$

So WFI for stream i is bounded by $\frac{g}{r_i}$. Normalized WFI is bounded by g, the granularity of rate allocation.

9 Acknowledgment

Special thanks go to Ritesh Ahuja, who showed interest in discussing RRR and helped this work by giving several useful comments. Authors are also indebted to Muddu Sudhakar, Venky and Sunder Rathnavelu for taking part in the initial discussions. We would also like to thank Dr. Huzur Saran of Computer Science and Engineering department of IIT-Delhi who was involved in discussions which generated some of ideas presented in this paper.

Thanks to Dr. Vijay Kumar of Bell Laboratories, who initiated this work by giving the first author of this paper, an opportunity to work in High Speed Networks Research Department. Thanks to the Computer Science Department of IIT-Delhi, where one of the authors spent most of his time working on RRR scheduling algorithm. Had these opportunities not been provided, it would have been impossible to bring this piece of work to its current form.

References

[1] S. Floyd and V. Jacobson. Link sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[2] Rahul Garg. Characterization of video traffic. Technical Report TR-95-007, International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, CA 94704-1198, USA, 1995. ftp://ftp.icsi.berkeley.edu/pub/techreports/1995/tr-95-007.ps.gz.

[3] Charles R. Kalmanek, Hemant Kanakia, and Srinivasan Keshav. Rate controlled servers for very high-speed networks. In *IEEE Global Telecommunications Conference*, pages 300.3.1 – 300.3.9, San Diego, California, December 1990.

[4] S. E. Lindberger, K. Tidblom. Weighted fair queueing, a method to control integrated heterogeneous traffic streams, having different QoS demands. In *Twelfth Nordic Teletraffic Seminar NTS12 (VTT Symposium 154)*, pages 47–58, Finland Technical Research Centre, Finland, August 1995. Finland Technical Research Centre.

[5] S. Sidropoulos M. Katevenis and C. Courcoubeti. Weighted round robin cell multiplexing in a general purpose ATM switch chip. *IEEE Journal on Selected Areas of Communications*, 9:1265–1279, October 1991.

[6] S.S. Panwar, T.K. Philips, and M.S. Chen. Golden ratio scheduling for flow control with low buffer requirements. *IEEE Transactions on Communications*, 40(4):765–772, April 1992.

[7] Abhay Kumar J. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.* PhD dissertation, Massachusetts Institute of Technology, February 1992.

[8] Dimitrios Stiliadis. *Traffic Scheduling in Packet-Switched Networks: Analysis, Design and Implementation.* PhD dissertation, University of California Santa Cruz, June 1996.

[9] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. In *Proceedings of INFOCOMM*, pages 111–119, San Fransisco, California, March 1996.

[10] Anujan Varma and Dimitrios Stiliadis. Hardware implementation of fair queueing algorithms for asynchronous transfer mode netwotks. *IEEE Communications Magzine*, pages 54–68, December 1997.

[11] Hui Zhang. *Service desciplines for packet-switching integrated-services.* PhD dissertation, University of California at Berkeley, 1993.

[12] Hui Zhang and Jon C. R. Bennett. Hierarchical packet fair queueing algorithms. In *Proceedings of ACM SIGCOMM*, pages 143–156. ACM, September 1997.

[13] Lixia Zhang. Virtual clock : A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9:101–124, May 1991.