10 Acknowledgment

We would like to thank Adisak Mekkittikul for help and numerous discussions about the scheduler design, particularly with the pipelining technique described on page 2.

11 References

[1] http://tiny-tera.stanford.edu/tiny-tera

[2] N.McKeown, “Scheduling Cells in an input-queued switch,” *PhD Thesis*, University of California at Berkeley, May 1995.

[3] N.McKeown et al. “The *Tiny Tera*: A small high-bandwidth packet switch core,” *IEEE Micro*, Jan-Feb 1997.

[4] N.McKeown, “Fast Switched Backplane for a Gigabit Switched Router”, *Cisco Systems white paper*, Nov 1997.

[5] R. Ahuja et al. “Multicast Scheduling for Input-Queued Switches,” *IEEE JSAC*, June 1997.

[6] Y. Tamir; G. Frazier, “High Performance Multiqueue Buffers for VLSI Communication Switches,” *Proc. of 15th Annual Symposium on Computer Architecture*, June 1988.

.

Table 2: Times (in nanoseconds) for each arbiter design.

Design	*N*=8	*N*=16	*N*=32	*N*=64
RIPPLE	3.72	4.38	6.18	4.95
CLA	2.53	2.27	4.4	9.5
EXH	1.11	1.66	2.48	4.44
SHFT_ENC	2.13	2.88	3.86	4.66
PROPOSED	1.01	1.46	1.64	2.3
% Improvement of PROPOSED over 2nd fastest design.	9.1%	12.0%	33.9%	48.2%

Table 3: Area of each arbiter design in 2-gate equivalents.

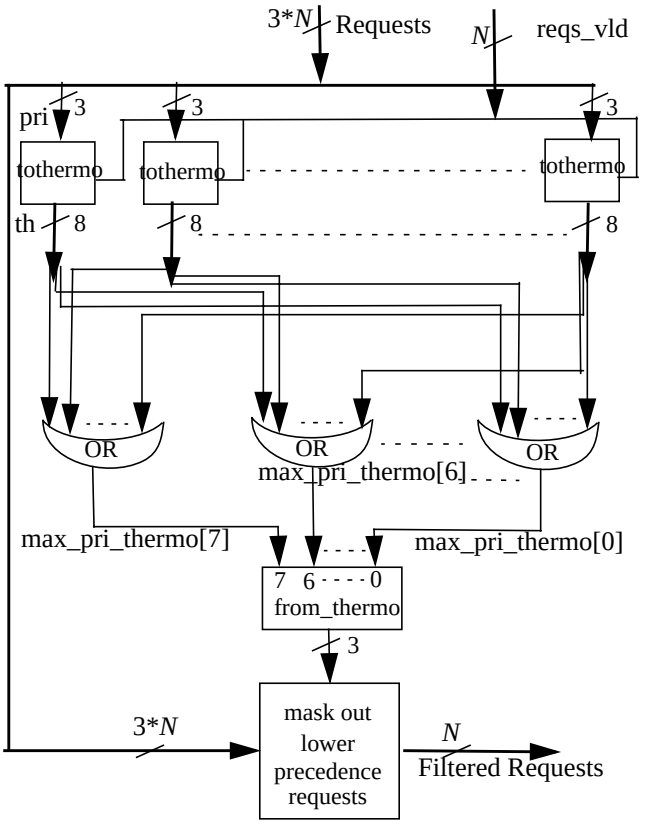
Design	*N*=8	*N*=16	*N*=32	*N*=64
RIPPLE	95	259	699	1132
CLA	96	191	500	721
EXH	351	1118	3760	13019
SHFT_ENC	296	985	2564	5085
PROPOSED	181	461	918	1390

8 Supporting Multiple Precedence Levels

Recall that the ESLIP algorithm used in the *Tiny Tera* supports four different strict precedence levels. i.e. each unicast and multicast request may take on one of four different precedence values. Because a precedence level takes strict priority over lower precedence values, the requests are pre-screened before being sent to the arbiter. All but the highest precedence requests are removed by the "precedence filter". The highest precedence request value is determined from the 2-bit precedence value associated with every unicast and multicast request.† Because the precedence filter sits in series with the arbiter, it also sits on the critical path of the scheduler. Therefore, we must design it carefully.

The precedence filter takes in *N* requests (each request is a 4-bit quantity: 2-bit precedence, 1-bit *vld*, 1-bit unicast or multicast) and outputs the *N filtered_reqs*. The functionality is equivalent to first calculating the highest precedence of all valid requests, and then zeroing out all requests which are not of that highest precedence value. Implemented naively, this could use a maximum-value function; an inherently 'ripple-like' function. Instead, our design (Figure 12) makes use of the thermometer encoding described in Section 6.3. After taking into account the global unicast-multicast mode bit, the four unicast and four multicast requests are first arranged into eight different precedence levels. This 3-bit precedence *pri[i][2:0]* is converted to a thermometer value *th[i][7:0]* of 8-bits. The 8-bit *max_pri_thermo* is then calculated by *max_pri_thermo[j]* = OR(*th[i][j]*) for all *i*=0...*N-1*. A reverse-thermometer transformation is then carried out to yield the highest precedence value, *max_pri[2:0]* which in turn is used to filter the original requests to yield *filtered_reqs*. As can be seen, we have used thermometer encoding to eliminate the ripple-like delay that would have been associated with a function that calculates the maximum of *N* 3-bit numbers.

Figure 12 Design of precedence filter



9 Conclusions

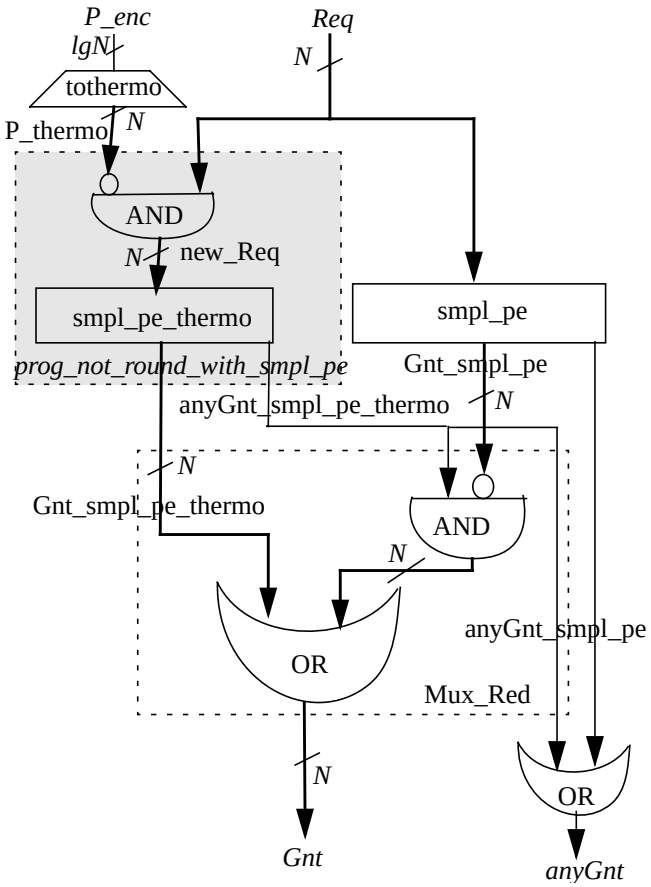
We conclude with what we think are the main contributions of this paper: (a) Request-grant and accept phases can be pipelined across different iterations to save precious clock cycles, and (b) Programmable priority encoders can be speeded up to be almost exactly as fast as static priority encoders, resulting in fast arbiters. In summary, we have shown that fair and efficient arbitration and crossbar configuration is feasible in hardware at very high speeds.

† At any given precedence, unicast and multicast take precedence over each in alternate time slots.

6.3 Preprocessing the input vector *Req* to eliminate programmability

Instead of decoding *P_enc*, we transform it to a thermometer encoded signal *P_thermo*. Now the exact same function as a *ppe_not_round* (Section 6.1) can be obtained by first ANDing $\overline{P\_thermo[i]}$ with *Req[i]* to obtain *new_Req[i]* and then giving *new_Req* as an input vector to another instance of a simple priority encoder (we will call this instance of a simple priority encoder as *smpl_pe_thermo*). This way we have succeeded in eliminating the programmability in the design. The complete design of this PPE (called PPE_only_smpls) is illustrated in Figure 11. Comparing the two PPEs shown in Figure 8 and Figure 11, note that the only difference is in the region marked by the shaded box in Figure 11. We will call the portion of the design inside the shaded box *prog_not_round_with_smpl_pe*.

Figure 11 Proposed design PPE_only_smpls



6.4 Recursive decomposition of PPE for large *N*

In this section we describe a technique for dividing a PPE into smaller sub-designs. This is useful, for instance, in round-robin arbiters that must arbitrate among a large number of requestors. With *PPE_only_smpls* shown in Figure 11, decomposition comes with virtually no overhead, and can even be used to further speed up the design for a particular value of *N*. The decomposition is possible only because the feedback loop has been eliminated (as described in Section 6.1)

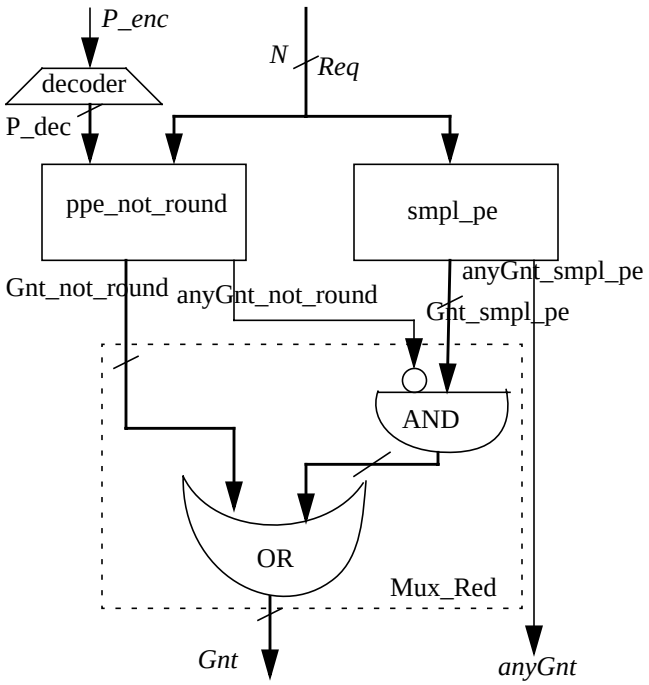
Decomposing into two levels of hierarchy, an *N*-bit PPE has two identical *N*/2-bit sub-blocks (we call each part PPE_rec_N/2). One sub-block considers inputs 0..*N*/2-1while the other considers *N*/2..*N-1*. Each sub-block contains one *smpl_PE* and one *prog_not_round_with_smpl_pe* (this uses the *N-1* least significant bits of thermometer encoding of *P_enc*). The output of the original *N*-bit PPE can then be obtained by suitably combining the sub-blocks, using the most significant bit of *P_enc* to choose which of the two parts will have a higher precedence. This idea can obviously be extended to further subdivide the *N*/2 bit sub-blocks into two *N*/4-bit sub-blocks each. Notice that there is little addition to the total area consumed with each level of decomposition, as the extra area needed to combine the outputs of the sub-blocks is offset by the decreased areas of the individual sub-blocks (decrease in areas is better than linear). There is also a trade-off in terms of speed, as each level of decomposition will add one stage of multiplexer (to combine the final outputs) while decreasing the delay through the sub-blocks. The exact level of decomposition will depend on the area and speed goals of the designer, and is obtained by experimenting with different designs.

7 Area and Timing Results

Table 2 shows the timing results of the various designs investigated in the preceding sections (see Table 3 for the area comparison). The CLA (carry lookahead) design in the table employs a lookahead of 2 for *N*=4, and a lookahead of *N*/4 for every other *N*. Design “PROPOSED” is the one obtained by implementing our two techniques mentioned above (in Section 6.1 and Section 6.2) and carrying out one level of the recursive decomposition mentioned in Section 6.4. All timings are in nanoseconds and all areas in 2-input gate equivalents. We believe that the reason the RIPPLE timing for *N*=64 is strange is because of the way the optimization tool chose to break the combinational feedback loop. (This shows how unpredictable designs can be with combinational feedback loops).

The designs were optimized under the same operating conditions with similar area and timing constraints using Texas Instruments’ TSC5000 libraries.The tool was requested to achieve the fastest implementation of each design. Of course, the numbers will depend on the standard cell library used, but these results are representative of the improvements possible with our techniques. The last row of Table 2 shows the percentage improvement in timing of the “PROPOSED” design when compared against the best times from more conventional arbiter designs (which happen to come from design EXH for all *N*).

Figure 8 Design PPE_comb



It works as follows : if there are no requests in the range *P_enc..N-1*, the output of the PPE is the same as the output of a simple priority encoder *smpl_pe*. If there is a request in the range *P_enc..N-1*, then the output of the PPE equals the output of *ppe_not_round*. This observation implies a simple 2:1 multiplexer for each of the *N*-bits of the output vector *Gnt*, with *anyGnt_not_round* as the select signal. The dashed box Mux_Red in Figure 8 is just a reduced multiplexer, equivalent to a multiplexer utilizing the fact that when *anyGnt_not_round* is '0', *Gnt_not_round* will be a zero vector, and so can directly by ORed to give the final output. This reduces the loading on the select signal (*anyGnt_not_round*) by half.

Design PPE_comb does not have any combinational feedback loop and can be fully optimized and tested by automated tools.

6.2 Eliminating the programmability in the PPE

Our second technique is to pre-process the inputs so we can create a PPE using only simple priority encoders. We first look at *thermometer* encoding, which we use in our design.

Thermometer encoding.

A thermometer encoding of a $\log_2(N)$-bit wide vector *x*, (i.e. $0 \le value(x) < N$) is a *N*-bit wide vector *y* with the following transformation equation:

$$y[i] = 1 \quad iff \quad (i < \text{value}(x)), \forall\ 0 \le i \le N$$

The truth table for *N*=8 is shown in Table 1 as one example of the transformation.

TABLE 1.

x(2..0)	y(7..0)
000	00000000
001	00000001
010	00000011
011	00000111
100	00001111
101	00011111
110	00111111
111	01111111

We find that this transformation take no longer than decoding *x* to give a 1-hot vector *x_dec*. For example, Figure 9 shows the equations for *N*=8. A fairly simple recursive algorithm can generate the equations for any *N* (that is a power of 2). Such an algorithm is sketched out in Figure 11. As can be easily seen from the description of the algorithm, each *y[i]* is either an OR or AND of no more than "width=$\log_2(N)$" terms. This is the same complexity as decode logic.†

Figure 9 Obtaining thermometer encoding for *N* = 8.

```
y7 = 0
y6 = x2.x1.x0
y5 = x2.x1
y4 = x2.(x1+x0)
y3 = x2
y2 = x2+x1.x0
y1 = x2+x1
y0 = x2+x1+x0
```

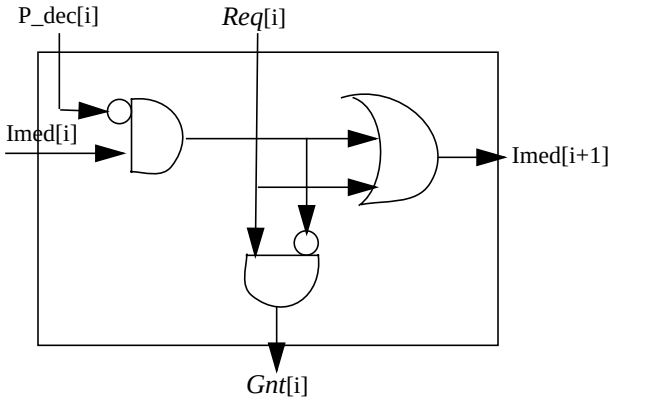
Figure 10 Obtaining thermometer-encoding for $N = 2^{width}$.

```
pow2 := 1;
y[0] := '0';
for i in 0 to width loop
  for j in 0 to pow2-1 loop
    tmp := y[j];
    y[j] := tmp OR x[i];
    y[j+pow2] := tmp AND x[i];
  end loop;
 pow2 := pow2+pow2;
end loop;
```

† In fact, it should be even simpler because decoding is a non-monotonic function which requires both inverted and non-inverted inputs to be available, unlike the thermometer transformation.

P_dec is obtained by decoding the input signal *P_enc*.The 1-bit signal *Imed[i]* is used to indicate whether the search process has already found a '1' (in which case *Req*[i] is to be ignored, *Gnt*[i] is '0' and *Imed[i+1]* is '1'). *Imed[i]* is to be ignored if *P_dec[i]* is '1' (which means that this is the "top priority request", and the starting point of the search process).

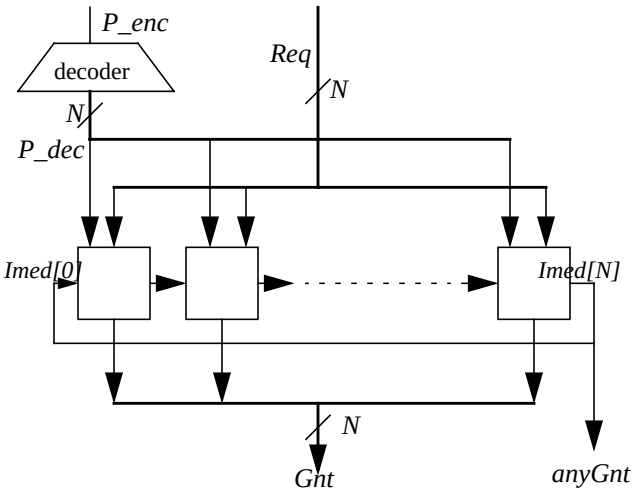
Figure 6 *mini_RPL*: a subblock of design RIPPLE



$Imed[i+1] = Imed[i].\overline{P\_dec[i]} + Req[i]$
$Gnt[i] = (\overline{Imed[i]} + P\_dec[i]).Req[i]$

Once we have the subblock *mini_RPL*, the RIPPLE PPE is simply a connection of *N* such sub-blocks connected back-to-back cyclically through *Imed*. (Figure 7)

Figure 7 Design RIPPLE



Clearly the design is area-efficient utilizing a small number of gates in each of *N* stages. However, the latency is large because the signal *Imed* has to ripple through *N* stages. This would be unacceptable in a high speed arbiter.

One way to speed up the RIPPLE design is to use carry-lookahead (we call this design CLA). For example, a carry lookahead of 16 would decrease the rippling delay to about 1/16th the original RIPPLE delay. We can trade-off area versus time in deciding the exact amount of lookahead. However, we have found that the resulting design has two fundamental problems:

First**,** there is a *combinational feedback loop* in the design. In fact, the loop is inherent to a RIPPLE-based PPE. Combinational feedbacks are undesirable because of several reasons: synthesis tools are generally unable to synthesize loops of logic, making it difficult to optimize the overall design. Also, most static timing analyzers are unable to work with combinational loops (unless the loop is broken, which then introduces significant inaccuracy in the results). These factors make such designs difficult to work with automated design tools.

Second, the design is slowed by the need to specify the highest priority input. This is evident in Figure 6 where we see that every sub-block *mini_RPL* takes *P_dec* as an input. Thus the critical path of *Imed[i]* to *Imed[i+1]* intrinsically consists of two "sub-stages" of delay within one mini-block stage.

6 An Improved PPE Design

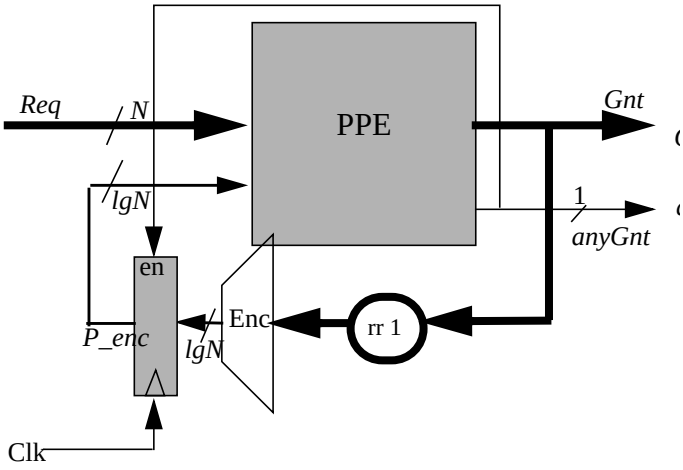
Our contribution consists of three ideas: the first two attempt to eliminate the combinational feedback loop and the long critical path caused by the programmable highest priority level; the third is a simple recursive decomposition allowing us to design large PPEs. The resulting design leads to improved area and speed characteristics.

6.1 Breaking the Combinational Feedback Loop

We can eliminate the combinational path by building the PPE from two simple sub-blocks. One is the simple non-programmable priority encoder: *smpl_pe* and a new block that we call *ppe_not_round*. *ppe_not_round* has the same inputs and outputs as a PPE, but searches for the first non-zero request after the current pointer value and *stops* the search process at the highest numbered input *N-1*. It does not cycle back to input 0, even if it couldn't find a request among the inputs *P_enc*...*N-1*. Therefore, *ppe_not_round* does not consider inputs *0*..*(P_enc-1)*. Note that *ppe_not_round* need not have a combinational feedback loop: we can construct it from a series of sub-blocks *mini_RPL* chained together through *Imed*, but without feeding *Imed[N]* back to *Imed[0]*. The usual carry lookahead techniques can be used to decrease the delay through the chain.

The new PPE (we call this design PPE_comb) is now constructed from a combination of a simple priority encoder and *ppe_not_round* as shown in Figure 8.

Figure 3 Block diagram of a round-robin arbiter



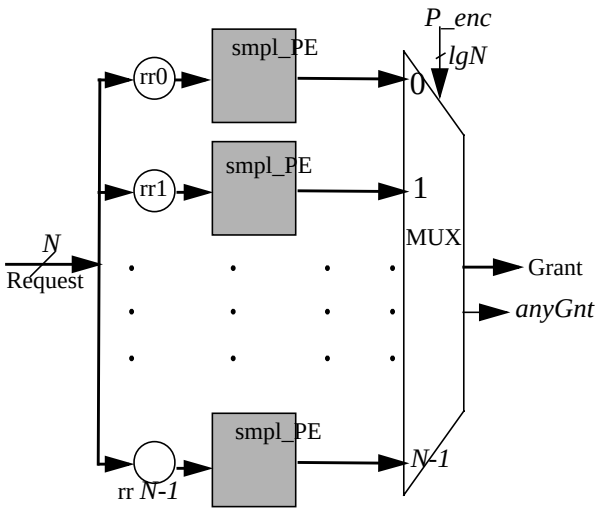
Note that the design of a fast non-programmable priority encoder (smpl_PE) is well known. It is the presence of programmability which complicates the design of our PPE.

We will first review some well-known PPE designs before presenting our PPE design. In each case, we will consider the following measures: (1) The area-speed complexity, and (2) How the complexity scales with *N* (the number of ports).

5.1 Design 1: Exhaustive (EXH)

Figure 4 shows the first PPE design. It uses *N* duplicate copies of a simple priority encoder smpl_PE, and the programmable priority input is used to select which priority encoder to use. The MUX shown is an *N*-to-1 (*N+1*)-bit multiplexer, with *P_enc* as the select signal. The circles marked as "rr i" denote wiring connections (there is no logic) such that *Req* is rotated 'i' positions to the right (most significant bit is the leftmost bit) before being presented to the multiplexer. Also *anyGnt* can equivalently be obtained by a simple *N*-bit OR of the inputs *Req* (the same observation holds for all designs). This design is reasonably fast for small *N*: an *N*-way MUX can be implemented in terms of a *lgN* stage 2:1 mux tree, with total time delay = t(smpl_PE) + *lgN* * t(2:1 mux) and total area=*N**area(smpl_PE)+area(*N*-way MUX). Clearly this design requires a large area of silicon, and is unsuitable for a large *N*. While the timing is expected to scale well, area consumption grows geometrically for increasing values of *N*. (more precise area and speed numbers for a particular technology are given later)

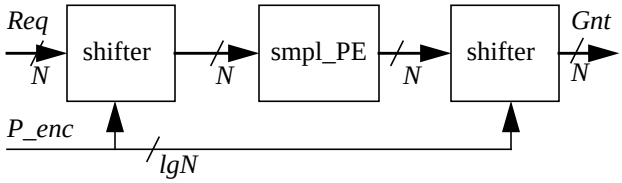
Figure 4 Design EXH



5.2 Design 2: SHFT_ENC

This is the most common design used in round-robin arbiters (Figure 5). It employs a barrel shifter to first rotate the incoming requests by *P_enc*, and one simple priority encoder whose output is again rotated by *P_enc* (this time in the other direction) to give the final output *Gnt*. Note that this design simply transfers the programmability to the shifters. An *N*-bit barrel shifter can be implemented by a series of *lgN* stages, stage *i* being a 2:1 mux for each of the *N* bits with *P_enc[i]* as the select signal. This design is lower speed than design EXH (as there are two shifters in the critical path) but much better in terms of area. Still the 2**lgN* stage delay through the shifters is expected to dominate the critical path, limiting the speed at which this PPE design can operate.

Figure 5 Design SHFT_ENC



5.3 Designs 3 and 4: RIPPLE and CLA

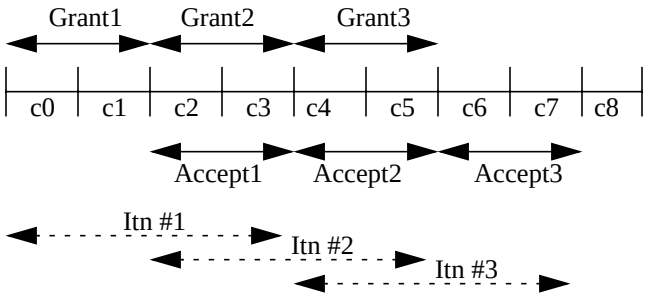
These two designs are both based on the following observation: the first non-zero request beyond the current pointer value can be found in a series of at most *N* sequential operations. First, we look at input *P_enc*: if *Req[P_enc]* is '1' then *Gnt[i]* will be '1' for *i=P_enc* and '0' otherwise. If on the other hand *Req[P_enc]* is '0', we look at the input numbered *(P_enc+1) (mod N)*, and continue this process till either we find a non-zero request or we are back to the input number we started from (*P_enc*). In the latter case, none of the input requests were '1', and so *Gnt[i]* will be '0' for all *i*, and *anyGnt* will be '0'. Each step can be carried out by a sub-block (that we call *mini_RPL*), as shown in Figure 6.

are combined in our implementation, Also shown in the figure is the decision feedback information from the accept arbiters, which is used in successive iterations to mask off requests from already matched inputs and outputs.

3 Pipelined Implementation

Our scheduler is designed to run at a clock speed of 175Mhz, and each time slot is composed of 9 clock cycles. It thus has approximately 51ns to complete three iterations of ESLIP. We found that a novel pipelining scheme allows the accept phase of one iteration to be overlapped with the request-grant phase of the next. As a result, *I* iterations require only *2I+2* clock cycles (instead of *4I* clock cycles without pipelining). With three iterations this saves 4 clock cycles.

Figure 2 The Grant-Accept Pipeline.



The pipelining scheme is shown in Figure 2. First, note that each phase consumes two clock cycles. The last clock cycle is used to update round-robin pointers and to prepare for the next time-slot.

At first glance, it appears that the three iterations of the scheduling algorithm need to be carried out sequentially. However, a simple observation allows us to start the grant phase of one iteration *before* completion of the accept phase of the previous iteration. Recall that before the grant phase of an iteration, we must mask off those requests that were accepted in the previous iteration. So we might expect that we must wait for the accept phase to complete to determine if a request should be considered in the next iterations. However, because we know that an input that receives at least one grant will definitely be matched, we can simply OR together all of its grants. This way we need not wait for the accept phase to complete before starting the next iteration. Of course, the accept phase must complete eventually to decide which input-output connection has been made.

4 Request Queues

Some request state needs to be kept inside the scheduler to tolerate the request-decision pipeline latency between the system linecards and the centralized scheduler chip. For each unicast virtual-output-queue (VOQ), we keep a 3 bit counter representing the number of requests pending in that corresponding queue. Also we keep a 5 element deep FIFO of pending multicast requests for each precedence. With each multicast request, we need to keep a 32-bit fanout. Thus the total amount of request state per input is 32*4*3 + 4*5*32 = 1024b. This was considered too much state to be kept in flip-flops. Neither did it make sense to put all of the state in memory: all the requests need to be visible at then start of each time slot. So instead, we keep most of the state in memory, and cache the head of the multicast FIFOs and one bit per unicast VOQ (indicating whether the VOQ is empty/non-empty) in flip-flops. This way we keep 32*4*1 + 4*32 = 256b in flip-flops, while the rest were stored in more dense on-chip memories. The total amount of request state in the scheduler chip was therefore 32 small memories and 8K bits in flip-flops. With certain amount of care in designing the logic of the cached bits, this results in substantial area savings on this chip.

5 Arbiters

The main contribution of this paper is the design of fast round-robin arbiters. As we see from the high-level block diagram in Figure 1, the delay through the grant and accept arbiters directly affects the speed of the scheduling algorithm. To make the arbiters fast, we first observe that a round-robin arbiter is equivalent to a programmable priority encoder, plus some state to store the round-robin pointer. A programmable priority encoder (PPE) differs from a simple priority encoder in that an external input dictates which input has the highest priority. In what follows, we take a detailed look at the design of a high speed round robin arbiter.

Figure 3 shows a block diagram of a round-robin arbiter. It has some state (called round-robin pointer, *P_enc*, of width *lgN* bits), which points to the current highest priority input. In every arbitration cycle, it uses this pointer *P_enc* to choose one among the *N* incoming requests, through a programmable priority-encoder. This PPE takes in *N* 1-bit wide requests and an *lgN*-bit wide pointer (which we call *P_enc*) as inputs. It then chooses the first non-zero request value beyond (and including) *Req[P_enc]*, resulting in an *N*-bit grant. Clearly, the core function of contention resolution is carried out by this combinational block. The pointer-update mechanism is generally simple and can be performed in parallel. To minimize overall delay, we focus on minimizing the path from *Req* to *Gnt*, which is a pure combinational path passing through the PPE. Hence, the problem of designing a fast round-robin arbiter is reduced to designing a fast PPE. It is to be noted that a fast PPE could be used in any arbiter, regardless of the pointer-update mechanism.

Design and Implementation of a Fast Crossbar Scheduler

Pankaj Gupta† and Nick McKeown
Depts. of Computer Science and Electrical Engineering
Stanford University, Stanford, CA 94305
{pankaj,nickm}@stanford.edu

† Currently affiliated with Cisco Systems, Inc., San Jose, CA 95134.

Abstract

Crossbar switches are frequently used as the internal switching fabric of high-performance network switches and routers. However, an intelligent centralized scheduler is needed to configure the crossbar fairly and with high utilization. In this paper, we describe the design and implementation of a scheduling algorithm for configuring crossbars in input-queued switches that support virtual output queues and multiple precedence (priority) levels of unicast and multicast traffic). This design was carried out for the *Tiny Tera* prototype at Stanford University[1][3], a fast label-swapping packet switch supporting 32 ports, each operating at 10Gb/s (OC192 line rate). The scheduler is designed to configure a 32×32 crossbar once every 51ns. The scheduler implements the ESLIP scheduling algorithm which consists of multiple round-robin arbiters.

1 The Scheduling Algorithm

The *Tiny Tera* has 32-ports operating at 10Gb/s and employs a (parallel) input-queued crossbar and a centralized scheduler [3]. Arriving variable length packets are segmented into fixed-size (64-byte) chunks which are switched every 51ns. During each 51ns time slot, the centralized scheduler considers the current occupancy of all of the input queues and calculates a new configuration for the crossbar. When the crossbar has been configured, each input delivers at most one (unicast or multicast) chunk into the crossbar fabric, and each output receives at most one (unicast or multicast) chunk. The crossbar is used to replicate multicast chunks when possible, and fanout-splitting is used to reduce head-of-line blocking [5]. The configuration must meet certain constraints: in each time slot, an input can be connected to at most one output for unicast traffic (but possibly several outputs for multicast traffic), and an output can be connected to at most one input for either unicast or multicast traffic. The *Tiny Tera* uses virtual output queues (VOQs) to eliminate head-of-line blocking [6] for unicast traffic and maintains four precedence classes. Thus there are 128 unicast (4 * 32) and 4 multicast queues at each input port; a total of 4,224 queues across all input ports.

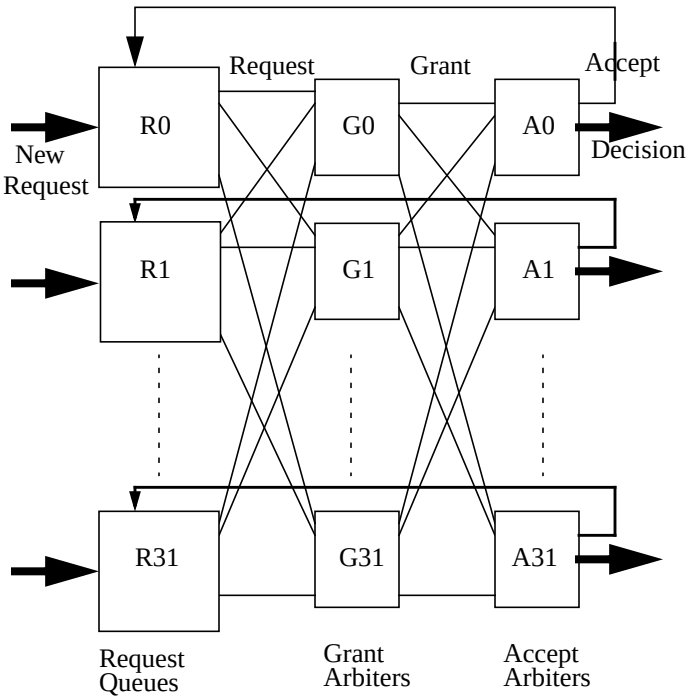
To support both unicast and multicast traffic, the *Tiny Tera* scheduler combines the *i*SLIP unicast scheduling algorithm[2] with the mRRM multicast scheduling algorithm[5]. The algorithm also supports four levels of precedence.‡ A detailed description of the ESLIP algorithm can be found in [4], and so we just give a brief overview of the algorithm here. Briefly: at the beginning of each time slot, the algorithm performs multiple iterations, each iteration consisting of three phases: (1) *Request*: Each input sends a request to every output for which it has a queued cell. (2) *Grant*: If an output receives any requests, it sends a grant to the one that appears next in a fixed, round-robin schedule starting from the highest priority element, and (3) *Accept*: If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. For brevity, we omit here the details on exactly how pointers are managed, and when they are updated. These details do not significantly affect the implementation.

‡ This is almost identical to the ESLIP algorithm used in the Cisco's 12000 GSR router.

2 High Level Design

A very high level block diagram of the scheduler is shown in Figure 1.

Figure 1 High Level Block Diagram of the Scheduler



The three phases (request-grant-accept) of the scheduling algorithm correspond to the three blocks shown in the figure. As the request phase of the algorithm just corresponds to forwarding of the requests to the grant arbiters, the request and the grant phases