Quality-of-Service for a High-Radix Switch

Nilmini Abeyratne, Supreet Jeloka, Yiping Kang,
David Blaauw, Ronald G. Dreslinski, Reetuparna Das and Trevor Mudge

University of Michigan, Ann Arbor, MI 48109

ABSTRACT

Communication in multi-processor systems-on-chip requires guaranteed throughput and latency. If the network is unaware of ongoing communication patterns, applications may not receive their necessary bandwidth or may suffer high network latencies. Many techniques have been proposed to provide quality-of-service (QoS) in the network by regulating network traffic; however, as network sizes have increased, the complexity of these techniques has grown as well, particularly in the case of multi-hop networks.

In this paper, we propose an efficient QoS implementation for a single-stage, high-radix switch, which is readily scalable to 64 nodes. In addition to best effort and guaranteed throughput services, we implement a guaranteed latency traffic class with a latency bound. Our implementation allows systems significantly larger than most current multi-core chips to be implemented without the need for difficult and complex multi-hop QoS.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors—*Interconnection architectures*

General Terms

Algorithms, Design

Keywords

network-on-chip, quality-of-service

1. INTRODUCTION

A system-on-chip (SoC) with real-time deadlines (e.g. a base station or an embedded system) consists of workloads that have both bandwidth requirements and latency constraints. The on-chip network in an SoC is shared among many nodes (cores/accelerators/IP blocks). Networks that are application-unaware will treat all messages equally even though some applications may require more bandwidth or lower latency for their messages. Quality-of-service (QoS) techniques can be added to networks to make them more application-aware. QoS techniques regulate access to a shared node, such as the memory controller, so that an application can meet its needs without degrading the performance of other applications.

Typically, QoS is implemented by controlling a variety of behaviors such as the injection time of packets into the network, total number of packets injected, and specific use of virtual channels and physical links [3, 6, 7, 8, 13, 18, 19]. As the number of nodes in systems continue to grow, a scalable interconnect becomes essential. Most recent work has focused on multi-hop network-on-chips (NoCs) to accommodate this growing number of nodes. Implementing QoS techniques in these NoCs, particularly to provide differentiated bandwidth and low latency services, is difficult. Some of these algorithms need information about traffic flows to be stored at each router in a multi-hop network, introducing large amounts of buffering.

Recently, the *Swizzle Switch* [15, 16], a high-radix crossbar that can provide a point-to-point interconnect fabric for many nodes, was introduced. While traditional crossbars cannot scale to higher radices due to power and area complexities, the *Swizzle Switch* was demonstrated to scale to a radix of 64 in 32nm technology at 1.5GHz [16]. This suggests that it can be used to build high-radix, single-crossbar networks. We will show that it is possible to implement QoS techniques within the *Swizzle Switch* with low complexity.

We augment the *Swizzle Switch* by implementing QoS with three traffic classes: Best-Effort (BE) class, Guaranteed Bandwidth (GB) class, and Guaranteed Latency (GL) class, in that order of increasing priority. The BE class is the default class in *Swizzle Switch* with least recently granted (LRG) arbitration [15]. To maintain the bandwidth requirements of the GB class, we derive a mechanism from Virtual Clock [19], a well-known QoS algorithm designed for packet switching networks. We efficiently integrate the Virtual Clock algorithm into the *Swizzle Switch* crossbar fabric and perform switch arbitration in a single clock cycle. An undesirable characteristic of Virtual Clock is that it couples the network delay of a flow with the bandwidth rate reserved by that flow. Packets from a flow with a small reserved rate have high network delay on average. Our interpretation of the Virtual Clock algorithm in the *Swizzle Switch* improves upon the original Virtual Clock algorithm by providing low delays to small-bandwidth flows.[1] Finally, our GL class is envisioned for sending infrequent, time-critical messages, such as interrupts, that need to quickly pass through the network. It is given the highest priority in the network, but we put safeguards in place to prevent its abuse. We calculate an upper bound in latency for GL class packets.

2. MOTIVATION AND BACKGROUND

2.1 Single-Crossbar Networks

A single-crossbar network is a single centralized crossbar as the on-chip interconnect. Large crossbars were considered infeasible because their area and power grow quadratically with radix. However, the recent *Swizzle Switch* has shown that crossbars can be designed more efficiently with the help of advanced circuit technology [15]. The *Swizzle Switch* crossbar reuses the wires of the output data bus to also perform switch arbitration, hence saving both area and energy. Single-crossbar networks provides benefits such as reduced network latency, dedicated input and output channels for each core, and uniform cache accesses. Furthermore, adding support for QoS, which involves adding storage to track flow history and global knowledge of bandwidth capacity, can be more easily achieved in a single-switch than in a multi-hop network.

[1] A *flow* is a stream of packets that traverse the same route from a source to a destination.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

2.2 Quality-of-Service Algorithms

Several QoS approaches for networks have been proposed. Static approaches such as Weighted Round Robin (WRR) and Deficit Weighted Round Robin (DWRR) can provide strict bandwidth guarantees [17]. However, WRR and DWRR lead to network underutilization as they do not distribute leftover bandwidth equally to flows with excess data or to best-effort flows. Fair Queuing (FQ) and Weighted Fair Queuing (WFQ) emulate bit-by-bit round robin (BR) service [2, 5, 12]. They compute finish times for packets, which is the time that the packet would have been serviced had the server been doing BR. However, computing finish times in these schemes have $O(N)$ complexity. Globally Synchronized Frames (GSF) [8] is a frame-based approach that controls the number of packets injected into the network at the source. It requires a global barrier network across all nodes, which adds overhead and can be slow.

The QoS mechanism we chose to implement was inspired by the Virtual Clock algorithm. Virtual Clock emulates time division multiplexing (TDM). In a true TDM system, packets are serviced only in the time slots allocated to the source. If the source has no packets to send, that time slot is wasted and results in link underutilization. Unlike TDM, Virtual Clock makes efficient use of link capacity by redistributing idle time slots to sources with excess demand.

A snippet of the Virtual Clock algorithm from [19] is shown below. Each flow has its own virtual time space kept in a *VirtualClock*. Transmitting one packet marks the passing of one virtual time step, *Vtick*, which is the average arrival time between packets from a flow in real time clock ticks. If the flow sends packets according to its average rate, its *VirtualClock* should approximately equal the real time clock. In the presence of multiple flows, the *VirtualClock*s of all flows are multiplexed to emulate a TDM system. Packets are stamped with the *VirtualClock* value of their flows at the time of their arrival and are transmitted according to increasing time stamps. If one flow does not send any packets for a while, its *VirtualClock* will fall behind. Then with a sudden burst, that flow can starve other flows until its *VirtualClock* value has caught up. To prevent flows from building up priority this way, their *VirtualClock*s are set to at least the real time clock, as shown in step 1 in the algorithm below. Now, any burst of packets will be interleaved with packets from other flows.

Original Virtual Clock Algorithm from [19]

- Upon receiving each packet from $flow_i$,
  1. $auxVC \leftarrow \max(auxVC, real\_time)$
  2. $auxVC \leftarrow (auxVC + Vtick_i)$
  3. stamp the packet with the $auxVC$ value
- Transmit packets in the order of increasing stamp values.

Coupling of Bandwidth and Latency: A drawback of the Virtual Clock algorithm is that it couples the reserved rate of bandwidth to latency. Flows with high reserved rates observe low average latency while flows with low reserved rates observe high average latency. The reason is that flows with high reserved rates have more packets to send; therefore, the switch must schedule these flows for data transmission more frequently. This results in very low average latency for packets. On the other hand, flows with low reserved rates are scheduled less frequently, resulting in high average latency for packets.

Previous work in the *Swizzle Switch* [14] had implemented a 4-level message-based QoS arbitration scheme. Our implementation has three main differences from the 4-level QoS implementation [14]. First, we allocate certain fractions of bandwidth to each input and ensure that they receive at least that much bandwidth. In the previous design inputs could only assign a priority level to messages and could not control how much bandwidth each priority level receives. Second, the previous design used a fixed-priority QoS mechanism (highest level messages are prioritized first), which could lead to starvation of messages in other levels. Third, the previous design required two arbitration cycles, whereas our entire arbitration (Virtual Clock arbitration + LRG arbitration) is within a single cycle. This is one of the new contributions of our work.

3. SWIZZLE SWITCH WITH QOS

In this work, we implement QoS in a single-crossbar network by extending the *Swizzle Switch* architecture to support three different traffic classes. We briefly explain each class and their priorities below.

Best-Effort (BE) class is applicable to flows that require neither guaranteed bandwidth nor guaranteed latency services. This class has the lowest priority in the network.

Guaranteed Bandwidth (GB) class is applicable to flows that require a guaranteed bandwidth from the network but not a tight bound on network latency; therefore, this traffic class has the second highest priority in the network. Bandwidth guarantees are maintained using the Virtual Clock algorithm.

Guaranteed Latency (GL) class is intended for time-critical packets that need to quickly traverse the network, such as interrupts or watchdog timers. The GL class is maintained by giving it the highest priority in the system. GL packets are serviced before any GB packets and may cause a disruption in GB services. To ensure that the GL class does not completely deny service to the GB class, a small fraction of bandwidth is reserved for it.

3.1 Swizzle Switch-Virtual Clock

The GB class is achieved by integrating the Virtual Clock algorithm into the *Swizzle Switch*, henceforth referred to as ***Swizzle Switch*-Virtual Clock (*SSVC*)**. In a *Swizzle Switch*, each crosspoint is configured to transmit packets of one particular flow, (In_i,Out_o). We added the following components to each crosspoint to support QoS:

- A virtual clock counter ($auxVC$)
- Virtual clock thermometer code register
- Virtual clock increment value register ($Vtick$)
- Replicated LRG arbitration logic

The virtual clock counter, $auxVC$, tracks the bandwidth usage history of the flow and is incremented by $Vtick$ each time a packet is transmitted. During arbitration, $auxVC$ values of requesting inputs are compared. We modified the switch arbitration circuitry to enable this comparison. The modifications are explained in detail below and also in Figure 1 and Figure 2. In Figure 1, five inputs of an 8-input switch is shown to be requesting some output M. To determine the winner, each input's $auxVC$ counters are used as the priority values in an inhibit-based arbitration.

***Swizzle Switch*'s Inhibit-Based Arbitration:** The *Swizzle Switch* employs an inhibit-based arbitration mechanism. In the beginning of the arbitration cycle, a subset of the output bus' bitlines, which are repurposed to perform switch arbitration, are pre-charged. During the arbitration, requesting inputs with higher priorities discharge the bitlines that they have priority over to inhibit inputs of lower priorities. At the end of the arbitration cycle, each input senses just one wire (the wire in the column with a '×' in Figure 1(c)). If other inputs have a priority bit '1' associated with this wire, then the '×' wire had been discharged and this input loses arbitration. Only a single input will remain with a still charged wire; that input had the highest priority amongst

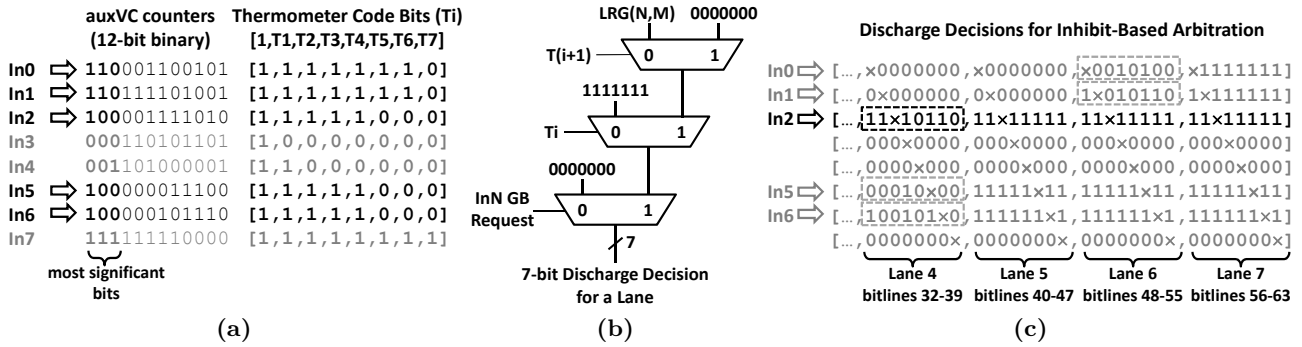

Figure 1: Example of a QoS-facilitated arbitration to some Output M. This example pertains to an 8-input switch with 64-bit output channels. (a) Five inputs are requesting Output M. Each (InN,OutM) crosspoint contains a 12-bit $auxVC$ counter, from which three most significant bits are used to create a thermometer code bit vector. The circuit in (b) uses two adjacent thermometer code bits to make discharge decisions for every lane (which, in this example, is a set of 8 bitlines). (c) Discharge decisions are then mapped to the output channel's bitlines for inhibit-based arbitration.

the requesting inputs and is the winner of the arbitration. The priorities can be updated in different ways to implement different arbitration policies [14]. We use least recently granted (LRG) arbitration in $SSVC$.

Thermometer Code Creation: Before $auxVC$ values of requesting inputs can be compared with inhibit-based arbitration, they must be mapped to the output bus' bitlines. The $auxVC$ counters (as seen in Figure 1(a)) are too large to be directly mapped to a typical 64-bit or 128-bit data bus. Therefore, we use some of the most significant bits of the $auxVC$ value and create a thermometer code vector. Ties between identical thermometer codes are broken with LRG arbitration. The use of LRG arbitration creates a restriction on the size of the thermometer code vector because the number of bitlines required for each LRG arbitration equals the number of inputs. The thermometer code vector is updated by shifting it up by 1 each time the most significant bits of $auxVC$ change (Figure 2).

Modified Inhibit-Based Arbitration: The $SSVC$ arbitration has two major components: 1) inputs with the smallest thermometer code bit vector must defeat all inputs with larger thermometer code bit vectors because in Virtual Clock the smaller the auxVC the higher the priority; and 2) ties between any inputs with the same thermometer code bit vector must be resolved with LRG. To achieve both components of the $SSVC$ arbitration in a single clock cycle, we created the small circuit in Figure 1(b). This circuit uses two adjacent thermometer code bits to determine discharge decisions for every set of 8 bitlines (or $lane$[2]). For example, to decide which bitlines to discharge in lane 4, the circuit in 1(b) uses thermometer code bits T4 and T5. This circuit is replicated for every lane, as can be seen in Figure 2. Finally, Figure 1(c) shows how these decisions are mapped to the output channel's bitlines. During the *Swizzle Switch*'s inhibit-based arbitration, each input will discharge the bitline where a '1' appears in 1(c). At the end of the arbitration, a single input will remain with its bitline still charged and that is the winner of the $SSVC$ arbitration.

As for the example in Figure 1(c), In0 senses in lane 6 because the three most significant bits of its $auxVC$ counter is "110". In0 will sense that wire 48 (the wire with '×') has been discharged by In1, In2, In5, and In6 and will lose the arbitration. In1 discharges wire 48 because it has higher LRG priority. In2, In5, and In6 discharge wire 48 because they get all '1's from the discharge decision circuit as they have a lower $auxVC$ value ("100"). Similarly, In1 will sense wire 49 and also lose to In2, In5, and In6. Between In2, In5, and In6, LRG in lane 4 picks the final winner. In2 will win because it senses wire 34, which is not discharged by any other input.

The accompanying Figure 2 shows the circuitry for $SSVC$. The most significant bits of the $auxVC$ counter has two purposes: 1) to determine the thermometer code bits and 2) to select the wire to be sensed by the sense amp. The wires being sensed can only be discharged by other inputs.

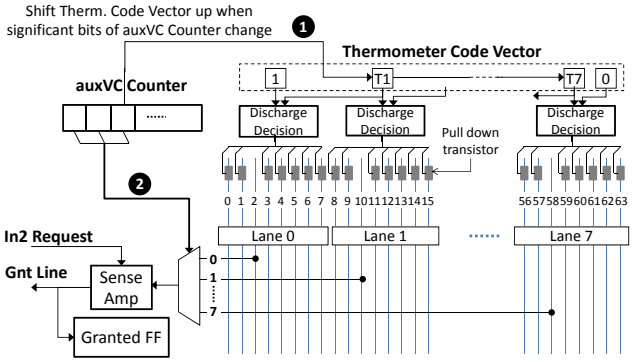

Figure 2: Circuitry for the Inhibit-Based Arbitration at Crosspoint (In2,OutM). Only 8 wires are needed for LRG arbitration in an 8×8 switch. Each set of 8 wires constitutes a *lane*. The discharge decision for each bitline is selected by feeding the thermometer code bits into the circuit in Figure 1(b). For In2, the sense amp can sense wires 2, 10, 18,...,58 in a 64-bit bus. The exact lane to sense is determined by the $auxVC$ value.

Finite Counters and Real Time Clock: The $auxVC$ counter increases by $Vtick$ each time a packet is transmitted. To prevent the finite counters in hardware from saturating, we modified step 1 of the original algorithm to be:

$$auxVC \leftarrow max(auxVC, real\_time) - real\_time$$

Also, instead of subtracting $real\_time$ from each $auxVC$ at every packet transfer, we keep a separate real time clock counter of the same granularity as the least significant bits of auxVC. Once that counter saturates, we subtract 1 from the most significant bits' value and shift down all thermometer codes by 1 position.

[2]A *lane* has exactly the number of bitlines required to perform LRG arbitration; usually equal to the number of inputs.

Improving Latency Fairness: We explored two more ways of managing finite counters: halving and resetting. Both methods provide latency fairness across bandwidth allocations by somewhat decoupling latency from reserved bandwidth rates. The *halving method* divides all $auxVC$ registers by 2 when any one of them saturate. The $auxVC$ register is shifted down by 1 position and the top half of the thermometer code is copied to the bottom half and then reset. The *reset method* resets all $auxVC$ registers to 0 when any one of them saturate. All thermometer codes are also reset to zero. After implementing these two methods, we observed a further improvement in latency for flows with low reserved rates.

3.2 Guaranteed Latency Class

The *SSVC* mechanism is used only by the GB traffic class to enforce bandwidth usage of flows. It cannot be used for the time-critical packets in the GL class because it couples bandwidth and latency and provides very high latency to low bandwidth flows. Therefore, we decouple bandwidth from latency by giving the GL traffic class the highest priority in the system regardless of its bandwidth usage. Its arbitration takes place in a separate *GL Lane*, leaving one fewer lane for the GB class. Figure 1(b) was modified to Figure 3 to support the GL traffic class. First, all ongoing GB arbitration are made to lose. Second, LRG arbitration selects one input if there are several inputs sending a GL packet. At the input ports, GL class packets should be buffered separately from GB class packets. Additional modifications to the sense amp circuit will be required to correctly sense the *GL Lane*.

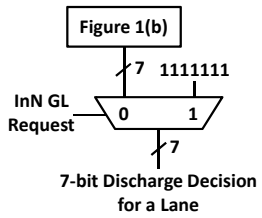

Figure 3: Modified discharge decisions circuitry to to support GL class arbitration. In the presence of a GL request, all bitlines in GB class lanes will be discharged.

3.3 Bandwidth Allocation To Traffic Classes

The BE class has no reserved bandwidth allocations and packets are serviced when neither GB nor GL packets are present. In the GB class, each individual input may request a fraction of the output channel's bandwidth; therefore, there can be as many GB flows per output as there are inputs. For the GL class, the output reserves a small fraction of bandwidth for any GL packet injected from any input to that output. Then, for each output channel, the sum of bandwidth allocated to all GB flows and the GL class should be less than or equal to the total bandwidth capacity of the output channel.

3.4 Guaranteed Latency Bound

As mentioned in Section 3.3, the output reserves a *small* fraction of bandwidth for any GL packet injected from any input to that output. A drawback of this approach is that multiple inputs may want to send GL packets, and because the GL class' bandwidth allocation is shared among all inputs, one input may take away from another's ability to send a GL packet within a reasonable network latency. Thus, our GL class is only applicable to types of time-critical messages that are very infrequent. Also, we only allocate a small fraction of bandwidth as the GL class has absolute priority over the GB class and may hinder the ability to maintain GB services. The bandwidth usage of the GL class is tracked by a counter similar to the auxVC counters of the GB class and increments by a tick count proportional to the reserved rate.

The following expression determines the maximum waiting time, τ_{GL}, for a buffered GL packet at the switch.

$$\tau_{GL} \leq l_{max} + N_{GL,o} \times (b + b/l_{min}) \quad (1)$$

l_{max} and l_{min} are the maximum and minimum length of a packet; $N_{GL,o}$ is the number of inputs injecting into the GL class to an output o; b is the buffer depth of GL class buffers. The term l_{max} accounts for the waiting time for channel release from a packet already holding the the channel. The term $(N_{GL,o} \times b)$ accounts for the transmit latency of buffered flits, and the term $(N_{GL,o} \times b/l_{min})$ accounts for the arbitration latency of each buffered GL packet.

In addition to knowing the worst case latency bound, it is also helpful to quantify the burst size allowed for some input requiring a latency bound for its packets. For example, if only one input is injecting to GL class and it expects a latency bound of 100 cycles, then it should not send more than 50 1-flit packets in a burst. But if there are 8 inputs and all of them expect a latency bound of 100 cycles, then each should not send more than 12 1-flit packets in a burst. We generalize this logic to two equations. First, we arrange all $N_{GL,o}$ inputs with GL packets to send in order from tightest to loosest latency constraint $\{L_1, L_2,...,L_N\}$. The maximum burst size in packets allowed for the input with the tightest latency constraint L_1 is

$$\sigma_1 = \frac{L_1 - l_{max}}{(l_{max} + 1) \times N_{GL,o}} \quad (2)$$

Recursively, the maximum burst size in packets for the input with the n^{th} tightest latency $(n > 1)$ constraint is

$$\sigma_n = \sigma_{n-1} + \frac{L_n - L_{n-1}}{(l_{max} + 1) \times (N_{GL,o} - n)} \quad (3)$$

The flow with the L_n latency constraint can burst as many flits as the flow with the L_{n-1} latency constraint but has to compete with the remaining $N_{GL,o} - n$ flows with higher latency constraints.

4. RESULTS

4.1 Evaluation Methodology

We wrote a custom, cycle-accurate simulator for the *Swizzle Switch* that modeled the *SSVC* mechanism in detail. To verify the correctness of *SSVC*, we further modeled the behavior of each wire, multiplexer, and sense amp in a C++ program. We tested this program with all input combinations of thermometer code vectors and valid LRG states. The arbitration decision of the level model was compared to the arbitration decision of a true (non-coarse grained) $auxVC$ value comparison to verify that each decision was correct.

The *Swizzle Switch* has been fabricated and tested in silicon [15] in 32nm industrial process. We have analyzed the impact on area and delay of the *Swizzle Switch* before and after adding the QoS logic. Wire delays were collected from SPICE modeling.

4.2 Evaluating Guaranteed Bandwidth

First, we demonstrate the ability of *SSVC* to adhere to reserved rates with Figure 4. Each input port has reserved a fraction of the output port's total bandwidth. We preallocated these fractions and pre-calculated each input flow's

Vtick. Unlike the LRG policy, which distributes bandwidth equally among inputs during congestion, the Virtual Clock policy distributes the bandwidth according to the requested rates, and guarantees that each input gets at least its requested rate. We simulated 200 combinations of reserved rates and a variety of packet sizes and verified that in each case *SSVC* is able to give flows their requested rates. Throughput loss from the *Swizzle Switch*'s arbitration cycle can be mitigated by applying techniques such as Packet Chaining [10] to multiple small packets headed to the same destination.

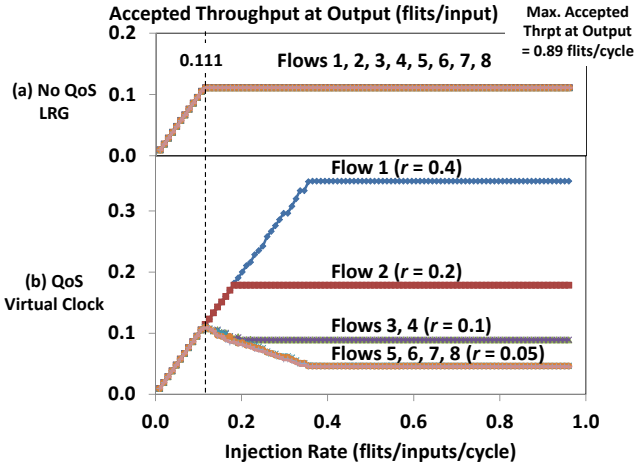

Figure 4: Bandwidth received by flows without and with QoS. (a) Without QoS, the switch performs LRG arbitration among the inputs. During congestion all flows receive an equal share of the available bandwidth. The maximum possible throughput is 0.89 flits/cycle because this experiment uses 8-flit packet sizes. (b) With QoS, all inputs get at least their reserved rate of bandwidth during congestion. The reserved fractions, r, of all 8 inputs are 40%, 20%, 10%, 10%, 5%, 5%, 5%, and 5%. *Details*: 8 inputs, 1 output, 128-bit output channel, 8-flit packets, 16-flit buffers, GB traffic only, 4 significant bits of $auxVC$ used for SSVC arbitration.

4.3 Improving Latency and Latency Fairness

Next we analyze the latency of GB packets with respect to the reserved rate of their flows. Figure 5 shows the average latency experienced by packets given the percentage of bandwidth allocated to a flow out of the output channel's total bandwidth capacity. The original Virtual Clock algorithm tends to give low bandwidth flows (<10%) very high latency. This is because flows with low reserved rates have fewer packets to transmit; therefore they are scheduled less frequently.

Our *SSVC* implementation greatly reduces the latency for smaller allocations because the comparison of $auxVC$ values is more coarse-grained. The LRG arbitration, which acts as a tie-breaker for multiple flows with the same $auxVC$ value, adds some fairness across the flows. However, the decrease in latency for smaller allocations comes with a sacrifice: the increase in latency for flows with larger allocations. We reason that this increased latency might not affect performance because nodes or applications which might need large allocations of bandwidth may have other built-in latency tolerance techniques, such as memory level parallelism.

The two other methods we explored for managing finite counters, halving or resetting the $auxVC$, further decreased the latency for flows with very low allocations (< 5%), especially during bursty injection. Results are shown in Figure 5. The insight is that by halving or resetting $auxVC$, we

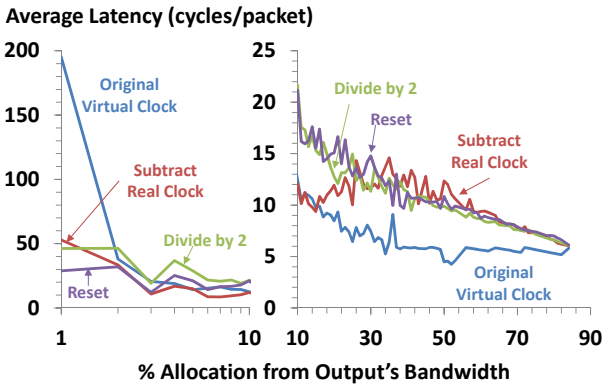

Figure 5: The *SSVC* implementation improved the packet latency for GB flows with low bandwidth allocations (<10%).

reduce the number of unique thermometer code values in existence, on average. During arbitration, contention between flows with the same thermometer code value is resolved using LRG, which introduces more fairness. As can be seen in the results, the reset to zero method has the least variance in latency across bandwidth allocations. All three methods were able to provide bandwidth to flows on average within −2% of their reserved rates.

4.4 Scalability

The accuracy of the *SSVC* technique increases with more "lanes" of arbitration. To support all three classes, at least three lanes are needed and each lane has to have as many wires as the number of input channels. However, the number of lanes are limited by the width of the output channel. The maximum possible number of lanes given a radix can be calculated by:

$$num\ \ lanes = \frac{output\ \ bus\ \ width}{radix}$$

For a radix-8, radix-16 and radix-32 switch, a 128-bit bus is sufficient. For a radix-64 switch, a 256-bit bus is required to support three QoS classes. Our QoS technique, while not scalable beyond 64 nodes, is sufficient to support applications in most modern SoC platforms. Scaling to more nodes involve composing multiple switches, which makes the QoS technique more complex. Crosspoints will have to be shared by several flows, requiring more per-flow state storage. In addition, composing multiple switches introduces conflicts in buffers at the input port. It becomes increasingly difficult to maintain separation between flows in buffers. However, the connectivity of 64 nodes is more than reasonable for current and near-term products.

4.5 Area and Delay

We calculated the storage overhead of *SSVC* for a crosspoint. Table 1 shows the storage overhead for a very large 64×64 switch with 512-bit channels is about 1MB. This is the worst case storage overhead.

The switch arbitration logic in the *Swizzle Switch* is located underneath the crosspoint on a separate metal layer. Without QoS support, the arbitration logic fits within the same area as the crosspoint width of a 128-bit channel. To determine the impact on area by QoS logic, we calculated the area consumed by major components of the Virtual Clock logic such as the $auxVC$ counters, the adder which increments $auxVC$ by $Vtick$, and the multiplexer before the sense amp that selects the lane to be sensed. We performed this calculation for several configurations of switch sizes (8×8,

Table 1: *SSVC* storage requirements (in bytes) for 64x64 switch with 512-bit output buses.

Buffering/ Input	BE	4 flits, 64 bytes/flit	256
	GB	4 flits/out, 64 outs, 64 bytes/flit	16,384
	GL	4 flits, 64 bytes/flit	256
Total buffering for all 64 inputs			1,056 K
Per-crosspoint state		auxVC(3+8 bits)	1.375
		Thermometer (8 bits)	1
		Vtick (8 bits)	1
		LRG (63 bits)	7.875
Total storage for 4096 crosspoints			45 K
Total switch storage (input port buffering + crosspoint storage)			1,101 K

16×16 and 32×32) and bus widths (128-bit, 256-bits, and 512-bits). With the *SSVC* logic, the crosspoint area for the 128-bit channel increased by 2%, which is equivalent to the area of a 131-bit channel. For 256-bit and 512-bit buses, the crosspoint area is large enough to comfortably house the *SSVC* logic without additional area overhead.

In addition, we calculated the impact on frequency. The critical path is extended by the multiplexer before the sense amp from Figure 2. The frequency slowdown is shown in Table 2. The worst slowdown is 8.4% for the 256-bit channel, 8×8 configuration.

Table 2: Frequency (in GHz) with and without *SSVC*.

	Channel Width					
	128		256		512	
Radix	*SS*	*SSVC*	*SS*	*SSVC*	*SS*	*SSVC*
8x8	4.17	3.95	3.47	3.18	2.52	2.35
16x16	3.44	3.15	2.50	2.35	1.62	1.55
32x32	2.45	2.30	1.60	1.54	0.94	0.92
64x64	-	-	0.93	0.93	0.51	0.51

5. RELATED WORK

In addition to the techniques we discussed in 2.2, there are several previously proposed techniques that offer support for GB and BE QoS classes. Æthereal [6] and Nostrum NoC [11] both use TDM and circuit switching to provide guaranteed bandwidth services. MANGO is a clockless NoC that provides GB using dedicated virtual channels [3]. A GB connection is established between two points in the network by reserving virtual channels along the path. LOFT combines locally synchronized frames and flit-reservation, which uses a look-ahead network to pre-schedule the data through each router [13].

Some techniques also provided explicit GL services. SonicsMX [18] has both priority threads and bandwidth threads, equivalent to our GL and GB classes, respectively. However, SonicsMX does not provide guarantees such as maximum network delay for its priority class. Credit-Controlled Static Priority (CCSP) decouples latency from the allocated bandwidth rate by using a scheduler that assigns a static priority among requesters [1].

Other works, such as QNoC [4] and CoQoS [9] implemented QoS management for multiple traffic classes based on system requirements.

6. CONCLUSION

In this paper, we proposed a QoS implementation for a single-stage, high-radix switch. We implemented three different traffic classes: Best-Effort (BE), Guaranteed Bandwidth (GB), and Guaranteed Latency (GL). We explained in detail the new *SSVC* circuit design for GB class that compares multiple priority values in a single clock cycle. The *SSVC* mechanism adds no more than 2% area-overhead to the *Swizzle Switch* crosspoint and incurs a frequency slowdown of at most 8.4%. Furthermore, our GL traffic class decouples network delay from reserved rate of bandwidth and offers low network latency to critical messages. We provided equations that accurately predict the maximum latency bound for GL class packets.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their input. This work was supported by DARPA under agreement #HR0011-13-2-0006 and ARM.

8. REFERENCES

[1] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *RTCSA*, 2008.

[2] J. Bennett and H. Zhang. Wf2q: worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[3] T. Bjerregaard and J. Sparso. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *DATE*, 2005.

[4] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Qnoc: Qos architecture and design process for network on chip. *Journal of Systems Architecture*, 50:105–128, 2004.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[6] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[7] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive virtual clock: A flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *MICRO-42*, 2009.

[8] J. W. Lee, M. C. Ng, and K. Asanović. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA-35*, 2008.

[9] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell. Coqos: Coordinating qos-aware shared resources in noc-based socs. *J. Parallel Distrib. Comput.*, 71(5):700–713, 2011.

[10] G. Michelogiannakis, N. Jiang, D. Becker, and W. Dally. Packet chaining: Efficient single-cycle allocation for on-chip networks. *Computer Architecture Letters*, 10(2):33–36, 2011.

[11] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *DATE*, 2004.

[12] J. Nagle. On packet switches with infinite storage. *Communications, IEEE Transactions on*, 35(4):435–438, 1987.

[13] J. Ouyang and Y. Xie. Loft: A high performance network-on-chip providing quality-of-service support. In *MICRO-43*, 2010.

[14] S. Satpathy, R. Das, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw. High radix self-arbitrating switch fabric with multiple arbitration schemes and quality of service. In *DAC-49*, 2012.

[15] S. Satpathy, K. Sewell, T. Manville, Y.-P. Chen, R. G. Dreslinski, D. Sylvester, T. N. Mudge, and D. Blaauw. A 4.5tb/s 3.4tb/s/w 64x64 switch fabric with self-updating least recently granted priority and quality of service arbitration in 45nm cmos. In *ISSCC*, 2012.

[16] K. Sewell, R. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. Wenisch, D. Sylvester, D. Blaauw, and T. Mudge. Swizzle-switch networks for many-core systems. In *JETCAS*, 2012.

[17] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, 1995.

[18] W.-D. Weber, J. Chou, I. Swarbrick, and D. Wingard. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *DATE*, 2005.

[19] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *SIGCOMM*, 1990.