# Using Dynamic Buffer Limiting to Protect Against Belligerent Flows in High-speed Networks

Fusun Ertemalp　　David R. Cheriton　　Andreas Bechtolsheim
Cisco Systems, Inc.　　(ertemalp, cheriton, avb)@cisco.com

## Abstract

*Conventional QoS mechanisms have focused primarily on providing better than normal service for some flows over others. With networks moving to much higher speeds and reasonable provisioning, best efforts access to network resources is adequate for common applications except when a* belligerent flow *attempts to consume an excessive amount of bandwidth. Any mechanism that attempts to contain such a flow must be able to operate at wirespeed in hardware. In this environment, conventional QoS mechanisms are not sufficient, either because they do not have mechanisms to contain these belligerent flows or because they are not practical to implement in hardware.*

*In this paper, we describe* Dynamic Buffer Limiting (DBL), *a buffer and queue management mechanism designed to recognize and handle belligerent flows at very high speed and suitable for hardware implementation.*

## 1. Introduction

Conventional quality of service (QoS) mechanisms have focused on providing preferential service for some designated flows over others and fair sharing among different competing applications. However, with the increasing bandwidth of networks, particularly enterprise and metropolitan-area networks, there are often adequate resources to meet the bandwidth and delay requirements of a wide range of applications. For instance, one voice flow at 32 Kbps uses only 0.03% of a 100 Mbps desktop link. 1000 voice flows use a total of 32 Mbps bandwidth, only 3% of a 1 Gbps link. Even video at 3 Mbps is a small percentage of a 100 Mbps Ethernet link. Commodity Ethernet switches and fiber optics make it inexpensive to provision networks with speeds of 100 Mbps or higher in the enterprise and metropolitan area.

In this environment, a major threat to adequate QoS for applications is misbehaving flows, what we call *belligerent* flows. Belligerent flows use as much bandwidth as available or excessive bandwidth compared to other flows, consume buffers, fill output queues, fail to honor reservations, and do not respond to congestion feedback, including RED-like packet drop congestion signaling. These flows are effectively "at war" with the network. Any sense of QoS is lost when these flows cause denial of service to other well-behaving flows.

Belligerent flows can arise from misbehaving or aggressive applications that try to use an excessive amount of bandwidth. An application without any congestion control mechanism that attempts to use all the available bandwidth would result in belligerent flows. An aggressive or improper TCP implementation on a high performance host can transmit high rate traffic to the network. Belligerent flows can also arise from network failures or misconfigurations. For instance, in a switched Ethernet environment, a spanning tree loop, where packets are forwarded at wire rate through a circle of switches[1] can result in several belligerent flows. Similarly, router misconfigurations can lead to looping behavior, such as multicast routing loops. A belligerent flow can also arise because of a host interface or operating system failure, where packets are transmitted into the network at its maximum rate. Less common but still of concern, a malicious host or compromised switch or router can inject high rate streams of packets into the network.

As networks move to higher speed, the need for a good protection mechanism increases because the higher speed allows belligerent flows to occur at higher speed as well. Then belligerent flows use up more resources more quickly.

Many of the scenarios above can result in very large numbers of belligerent flows. For instance, every new flow initiated during the time of a spanning tree loop can turn into a belligerent flow, potentially resulting in hundreds of belligerent flows over the 30 seconds or longer period that such a loop can persist. Moreover, containing one belligerent flow does not in general prevent additional belligerent flows from coming into existence.

Thus, with the mission-critical nature of most networks, a protection mechanism is required that is able to react quickly to and handle a large number of belligerent flows.

[1] We use the term *switch* to refer to a high-speed multi-layer enterprise switch/router that handles Ethernet and IP forwarding at wire-speed.

1092-1658/01 $17.00 © 2001 IEEE

Moreover, with 1.5 million packets per second (Mpps) packet input rate on every port in a gigabit Ethernet switch and 15 Mpps per port in a 10 Gbps switch, a hardware implementation is necessary. Finally, the protection mechanism should not interfere with the fair sharing and efficient utilization of the network in the absence of belligerent flows, providing as good a service as possible when belligerent flows are present.

In this paper, we describe *Dynamic Buffer Limiting (DBL)*, a QoS protection mechanism providing switch buffer and queue management that is implementable in high-speed switch hardware at a low cost per flow. DBL protects a switch and the flows passing through it from belligerent flows, reacting quickly to excessive buffer demands.

The next section describes the DBL mechanism in detail. The following sections describe simulation-based performance results, the cost of DBL and a comparison to related work. We close with conclusions.

## 2. Dynamic Buffer Limiting (DBL)

The basic *Dynamic Buffer Limiting (DBL)* mechanism operates by keeping track of the amount of buffering for each flow in the switch. On arrival of a packet, if the buffer usage of the flow to which the packet belongs exceeds a dynamically computed buffer limit, DBL marks the packet[2]. This marking signals the flow to adapt to the limited output bandwidth by slowing down, relying on TCP or TCP-like congestion control. If the flow fails to slow down sufficiently after some period, the switch concludes that the flow is non-adapting and classifies it as a belligerent flow. The switch severely constrains the buffering that a belligerent flow can use, typically dropping all subsequent packets of that flow until the flow's buffering falls below a much lower *belligerent flow buffer limit.* A flow can subsequently escape this belligerent flow classification by staying below this buffer limit across several packet receptions.

We next present the algorithm in detail and then describe how it handles belligerent flows.

### 2.1. DBL Algorithm

DBL maintains a flow table with an entry per flow, as illustrated in Figure 1. Each flow table entry consists of two fields: `buffersUsed` and `credits`. The `buffersUsed` is the number of packet buffer allocation units, *cells*, occupied by the packets of the flow that are currently stored in the output queue. The `credits` field records the history of the flow to distinguish between adaptive and non-adaptive flows. It keeps track of how much more buffering a flow uses after the congestion signals. There is a flow table per output queue. DBL also maintains, per output queue, the length of the queue in buffer units, `currentQueueLen`, and the number of active flows in the queue, `activeFlows`. An *active flow* is defined as a flow that has packets in the queue.

[2]The packet is marked by dropping it or by setting the Explicit Congestion Notification bit in the packet. We evaluate the DBL algorithm with dropping only.

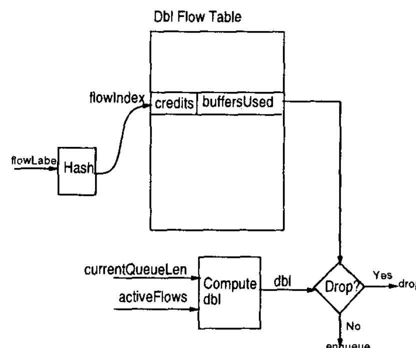


**Figure 1. DBL State and Operation Overview at an Output Queue**

Referring to Figure 1, on packet arrival, the packet is mapped to a flow table entry by hashing its flow label to an index into the table. For instance, an IP flow is hashed using the IP source and destination addresses, L4 port numbers (if TCP or UDP) and IP protocol type. The dynamic buffer limit, `dbl`, is computed (or looked up from a table) based on the `currentQueueLen` and `activeFlows` of the output queue, as suggested in bottom of Figure 1. `dbl` is limited at most to a `dblMax` value to prevent a single flow from using excessive queue space during periods of low demand and at least to a `dblMin` value to allow some minimum amount of buffering to each flow even under high loads.

According to the flow's buffersUsed and credits values and the output queue's dbl value, the packet is either dropped or enqueued to the output queue, as illustrated in the diamond at the bottom right in Figure 1.

The drop decision uses three global parameters: `bfCreditLimit`[3] is the number of credits below which a flow may be considered a belligerent flow depending on its buffer usage. `bfBufferLimit` is the maximum amount of buffering a flow is allowed once it is identified as a belligerent flow. Section 2.2 discusses the trade-offs in setting these parameters. `markProb` is the probability used in deciding to drop a packet and decrement credits when a flow's buffer usage is larger than `dbl`.

[3]`bf` is is used for belligerent flow.

If the flow's credits are less than or equal to `bfCreditLimit` and the flow's buffer usage is over `bf-BufferLimit`, the flow is treated as a belligerent flow. This indicates that the flow has not responded to the congestion signals and is continuing to use an excessive amount of buffering under congestion. The packet is dropped and the credits are decremented by one.

If the flow is not considered a belligerent flow, its buffer usage is greater than `dbl`, the packet is to be marked (probabilistically determined by the `markProb` value), and if the credit field is at the maximum, the packet is dropped. The credits field at maximum indicates that this is a first drop. Otherwise, the packet is enqueued. In either case, the credits are decremented. If the flow does not back-off in response to the single drop, the credits continue to decrement probabilistically with further packets, causing the flow to be considered a belligerent flow when it is below the `bf-CreditLimit`. Only a *single* packet is dropped to signal congestion so that TCP and TCP-like flows have a chance to back-off without losing more packets in their windows. Otherwise, potentially back-to-back packets from the TCP flow would be dropped, forcing TCP into slow-start and consequently bad performance.

If the buffer usage for the flow is less than `dbl`, the packet is enqueued to the output port. If the flow's credits are below `bfCreditLimit`, then its credits are incremented only by one because we still suspect the flow is a belligerent flow. Otherwise, the credits are reset to the maximum, because we no longer suspect that the flow is a belligerent flow. The `buffersUsed` of the flow and the `currentQueueLen` are both incremented by the number of cells the packet occupies. Table 1 summarizes the DBL packet and credit handling for different `buffer-sUsed` and `credits` ranges. In the table, prob. refers to probabilistic.

**Table 1. DBL decision matrix.**

| buffersUsed | credits | pktAction | credAction |
|---|---|---|---|
| $> bfBufferLimit$ | $<= bfCreditLimit$ | drop | decr |
| $> dbl$ | $== maxCredits$ | prob. drop | prob. decr |
| $> dbl$ | $< maxCredits$ | enqueue | decr |
| $<= dbl$ | $<= bfCreditLimit$ | enqueue | incr |
| $<= dbl$ | $> bfCreditLimit$ | enqueue | maxCredits |

On packet transmission, the buffer count in the flow table for this packet is decremented by the size of the packet in cells. Also, the `currentQueueLen` is reduced by the packet length in cells. The hash index to access the flow table on transmission can be obtained by recomputing it from the packet header or it can be stored in the packet descriptor, depending on the switch architecture.

## 2.2. DBL Belligerent Flow Control Parameters

DBL uses two global parameters in the drop decision, `bfCreditLimit` and `bfBufferLimit`, to identify and throttle belligerent flows. The setting of these parameters depends on trade-offs in belligerent versus non-belligerent flow handling.

A high value of `bfCreditLimit` means it is harder for a flow to escape the belligerent flow classification and a flow is classified as a belligerent flow sooner. This allows less burst time for a flow to back-off after a congestion signal. A low value of `bfCreditLimit` means it is easier for a flow to escape the belligerent flow classification. In the expected case, when a flow is reduced to `bfBufferLimit` because its credits fall below `bfCreditLimit`, its credits are typically further reduced to 0 in short order with further incoming packets from that flow. A high-speed flow that does not adapt at all typically stays at 0 credits. The flow has to transmit $bfCreditLimit+1$ consecutive packets through the switch without encountering drop, while operating under the belligerent flow limit of `bfBufferLimit`. We set the `bfCreditLimit` parameter to 2 so that a belligerent flow has to transmit 3 consecutive packets through the switch without encountering drop.

The `bfBufferLimit` is set to the number of cells in a maximum size packet to severely limit belligerent flows and to protect the switch and network as much as possible. Ideally, an application should detect overload and reduce the source packet rate rather than relying on the network to perform this task. One might argue for denying the belligerent flow because a belligerent flow can contribute to additional output delay and consume a significant amount of bandwidth (See Section 4). However, DBL does not completely deny a belligerent flow because of the danger of falsely classifying a TCP as a belligerent flow, the risk of hash collisions into the same bucket, and the desire to allow a TCP flow to recover from this classification. In particular, if a TCP flow fails to respond to a drop in time, it is forced under the belligerent flow limit. However, after encountering multiple drops, the TCP flow goes into retransmission timeout, allowing its buffers in the switch to drain, and falls below the belligerent flow buffer limit. It then resumes a slow ramp up of its congestion window from a low value, allowing it to escape from its belligerent flow classification.

## 2.3. Fast Belligerent Flow Classification/Throttle

Figure 2 illustrates the buffering and credit behavior for a belligerent flow as it is recognized and contained. The belligerent flow is a 200 Mbps flow going to a 100 Mbps port.

The flow starts with the maximum number of credits, `maxCredits`. When the buffersUsed goes over `dbl`, the

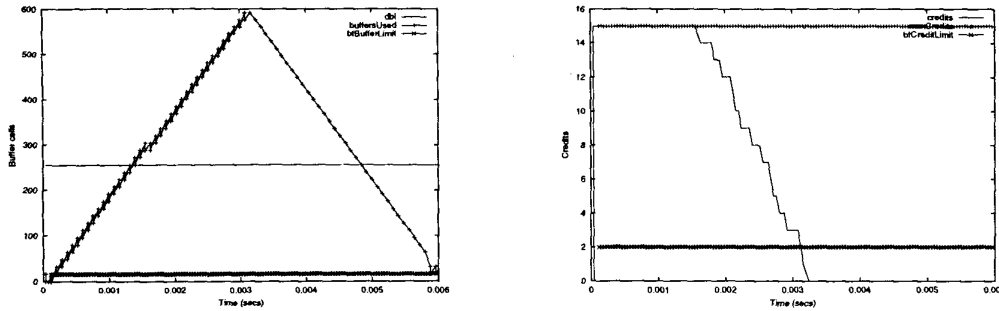


**Figure 2. DBL belligerent flow handling (a) Buffers and (b) Credits. In (a), dbl is the middle line, buffersUsed is the pyramid line and bfBufferLimit is the bottom line. In (b), the top line is maxCredits, the bottomline is bfCreditLimit and the dropping line indicates the number of credits for the flow over time.**

flow loses one packet but does not respond to the drop and continues to send packets at the same high rate. The flow then probabilistically loses its credits one by one, because it's buffer usage remains over dbl. At time 0.003 secs, it goes below `bfCreditLimit` and is dynamically classified as a belligerent flow. At this point, the flow's buffer usage is limited to `bfBufferLimit`, causing *all* new packets to be dropped and credits to go to zero at time 0.0033. At time 0.0058, the belligerent flow's buffer usage is reduced to `bfBufferLimit`, minimizing its effect on the switch resources. If and when the flow reduces its rate or when the congestion at the output port is gone so that the flow can enqueue $bfCreditLimit + 1$ packets to the output queue without drop, it escapes the belligerent flow classification and immediately regains maximum credits.

As this example illustrates, the belligerent flow is throttled back to a modest amount of buffering within a short period of time, namely 6 milliseconds. 6 milliseconds is equal to only 0.44% of 16 MB packet buffer time at 100Mbps. Thus, the impact of the belligerent flow on the switch in terms of buffering is short, and even shorter if the belligerent flow is higher rate.

The belligerent flow identification time depends on the contention in the queue, namely the number of active flows, the current queue length, the input rate of the flow and the drain rate of the flow. The *maximum* time to identify a belligerent flow is: $\frac{dbl+maxBufferAfterDbl}{flowInputRate-flowDrainRate}$. If the $flowInputRate$ is $n$ times the $flowDrainRate$, the above equation can be written as $\frac{n*(dbl+maxBufferAfterDbl)}{(n-1)*flowInputRate}$. $dbl$ is at most $dblMax$ but much lower if the queue is congested. The maximum additional buffer usage of a flow after the packet drop signal, $maxBufferAfterDbl$, is $(maxCredits - bfCreditLimit) * (\frac{1}{markProb} - 1) * maxPacketSize$. In our implementation with $maxCredits$ at 15 and a $bfCreditLimit$ of 2, the above difference is 13. A $markProb$ of .333 means decrementing the credits once every 3 packets allowing roughly $2 * 13 = 26$ packets before hard drop. With 1000 byte packets used in our simulations, this allows 26 kilobytes of buffering after the packet drop signal. $dblMax$ is picked as 16 kilobytes to allow for a typical NFS burst size. The selection of these parameters according to a representative enterprise network policy are omitted for lack of space but described in [4]. Figure 3 shows how the *maximum* belligerent flow identification time changes according to the ratio between flow input and drain rate for the above $dblMax$ and $maxBufferAfterDbl$ values when the flow input rate is 200, 400, 600, 800 or 1000 Mbps. When the congestion is higher, as indicated by the lower drain rate or when the belligerent flow arrives at a higher input rate, the belligerent flow is identified sooner.

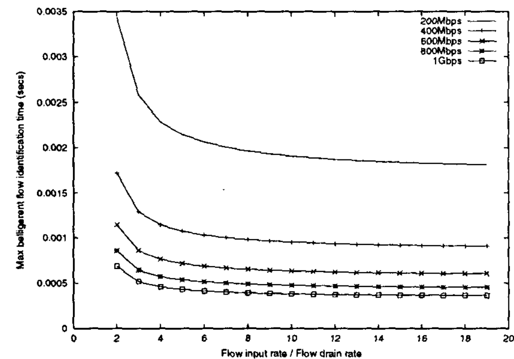


**Figure 3. DBL identifies belligerent flows quickly**

The time to throttle a belligerent flow to a few buffers de-

pends on the output rate of the queue and the current queue length. The assumed high transmission rate for the switch ports means the packets queued in the switch on behalf of a belligerent flow are transmitted out in a short period of time, bringing the flow's buffer usage quickly down to the `bfBufferLimit`. The fast reaction to a belligerent flow prevents it from consuming excessive buffers in the switch, allowing the switch to recover buffering from the belligerent flow quickly and continue to service normal traffic. Overall, DBL reacts quickly to belligerent flows.

DBL also provides fair sharing between robust TCP flows and protects low-rate, drop-sensitive fragile flows such as voice or telnet flows. The interaction of DBL with robust and fragile flows and related simulation results are discussed in detail in [4].

## 3. Hardware Implementation Cost

DBL has been realized on a high-speed switch ASIC, handling DBL processing at *wirespeed* on tens of gigabit Ethernet ports simultaneously, i.e. several tens of millions of packets per second.

The implementation incurs very low cost per flow, allowing it to handle a large number of flows. In particular, the flow table memory for all the ports in the switch is implemented in a single dual-ported external SRAM chip providing 512K flow entries of 18 bits each. With one bit used for memory parity, each flow entry has a 13-bit buffersUsed and a 4-bit credit field, allowing a `maxBuffersUsed` of $(2^{14} - 1)$ cells per flow (1MB of buffering with 64-byte cells) and a `maxCredits` value of 15.

The per-queue dbl state and dbl lookup table are small on-chip tables. The lookup table approach compared to hard-wiring the computation results in simple hardware logic and allows changing the dbl computation according to different network policies and requirements without ASIC changes. The DBL logic is easily incorporated into the normal packet reception and transmission pipeline, given DBL processing only needs to be done at packet enqueue and dequeue time. The on-chip gates required to implement DBL are less than 3 percent of the switch ASIC and the external SRAM is less than 1 percent of the total material cost of the system.

Assuming 25 percent hash table utilization to have low probability of hash collisions, this implementation can support roughly 128K flows with a cost of $4 * 18 = 72$ bits of dual-port off-chip SRAM per flow (The on-chip per-flow cost is negligible, roughly 2.8 bits.) 128K flows are more than the number of flows that would fit at once in the packet memory of the switch, so DBL state is not the limiting resource. Furthermore, an additional SRAM can be added to double the DBL flow table size, allowing use of the flow hash table with 12.5% utilization, reducing probability of hash collisions even further.

## 4. Performance Evaluation

We evaluated DBL protection against belligerent flows in high speed networks. We run simulations with a variety of flow types in the presence of one or more belligerent flows using the ns-2 simulator[11].[4] All the TCP sources are implemented using NewReno TCP. The long-lived TCP connections are modeled with file transfers. The belligerent flows are implemented using wire-speed UDP sources.

All the links in the simulation are 100 Mbps and have 550 nanoseconds of propagation delay. This is the same as the maximum delay in a 100 Mbps Ethernet Lan over Cat-5 UTP cable. All switch output queues have a limit of 1024 packets. A packet size of 1000 bytes is used. The DBL parameters are set according to the example network policy described in [4], with `dblMax` parameter set to 256 cells, or 16 kilobytes, `dblMin set to 32 cells` and `markProb` set to .333. DBL results are compared to RED, FRED and a perfect implementation of per-flow queuing FQ. We compare to RED, because RED is the widely used active queue management algorithm and is recommended by the Internet community [1]. We compare to FRED, because FRED combines RED with per flow isolation. RED and FRED are each configured with a minimum threshold of 64 packets, a maximum threshold of 192, a queue weight of 0.002, and a drop probability of 0.02.

### 4.1. Fragile Flow Protection Against Belligerent Flows

Figure 4 shows the throughput of 20 low rate flows each at 2 Mbps and one wire-speed belligerent flow passing through a 100 Mbps port. A new fragile flow is started at each 1 second interval.

In this experiment, the fragile flows do not suffer even a *single* packet drop. They are well-protected and they receive their 2 Mbps bandwidth, as indicated by the belligerent flow rate being reduced by 2 Mbps with the introduction of each new fragile flow. Thus, when all 20 fragile flows are active consuming 40 Mbps of the output link bandwidth, the belligerent flow is limited to the remaining 60 Mbps[5].

[4] All network protocols and RED queuing algorithm used in the simulation are part of the standard ns-2 distribution. We implemented DBL algorithm as described in Section 2 as a queue type in ns-2. The ns-2 FRED implementation is from the contribution of CSFQ work [13] to ns-2. FQ results are computed as $\frac{linkrate}{numberofflows}$

[5] This experiment also indicates the a belligerent flow can still be using a substantial amount of bandwidth in a network, just left-over bandwidth. This would be a problem if the network accounting is usage-based, but that is generally not the case in enterprise or metropolitan networks of interest.

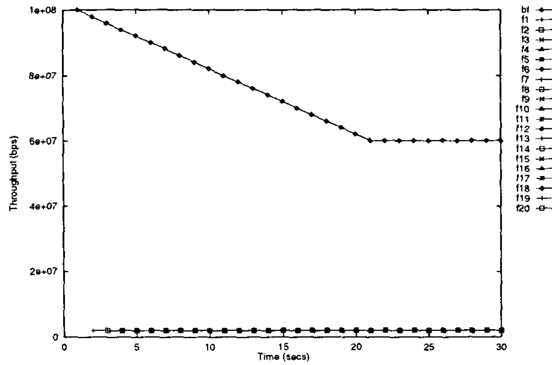


**Figure 4. DBL pulls down the belligerent flow to the remaining bandwidth from fragile flows**

Using RED in the same traffic scenario[6], the fragile flows experience between 5 and 23 percent packet drop, making them effectively unusable for applications such as voice, video and interactive sessions. Not surprisingly, DBL also provides lower mean delay per flow in this simulation, namely 1.0-1.2 milliseconds versus 10 milliseconds with RED.

Additional simulations were run with more fragile flows and more belligerent flows (not included for lack of space). DBL protected the fragile flows until the demand by the fragile flows exceeded the output link rate. Naturally, fragile flow drops also happen when the output queue becomes full.

### 4.2. TCP Flows with a Belligerent Flow

With 5 TCP flows and 1 wire-speed, 100 Mbps belligerent flow, this simulation illustrates how DBL limits a belligerent flow that shares a queue with robust flows. Table 2 shows the goodput of each flow.

**Table 2. Average Goodput (Mbps) per TCP and One Belligerent Flow**

| | tcp-0 | tcp-1 | tcp-2 | tcp-3 | tcp-4 | bf |
|---|---|---|---|---|---|---|
| DBL | 20.3 | 19.2 | 19.4 | 18.9 | 19.8 | 4.0 |
| FRED | 17.7 | 16.7 | 16.7 | 16.5 | 17.1 | 17.2 |
| RED | 2.7 | 1.7 | 1.6 | 1.2 | 0.8 | 94.1 |
| FQ | 16.7 | 16.7 | 16.7 | 16.7 | 16.7 | 16.7 |

As shown by the throughput of the belligerent flow in Table 2, DBL contains the belligerent flow and brings its data rate down to 4 Mbps. Limiting the belligerent flow to a low rate allows TCP flows to have high goodput. Moreover, the sharing among the TCP flows continues to be balanced. However, the belligerent flow is not completely denied, and it continues to use a significant amount of bandwidth.

[6] In this case, all flows started to transmit at the start of the test so that drop percentages can be compared relatively.

FRED similarly limits the belligerent flow, although less severely than DBL. In contrast, with RED, a single belligerent flow causes almost denial of service to TCP flows, consuming 94.9 percent of the output bandwidth.

### 4.3. TCP Flows and Multiple Belligerent Flows

Here, the previous scenario is extended with additional wire-speed belligerent flows arriving randomly in the first 5 seconds.

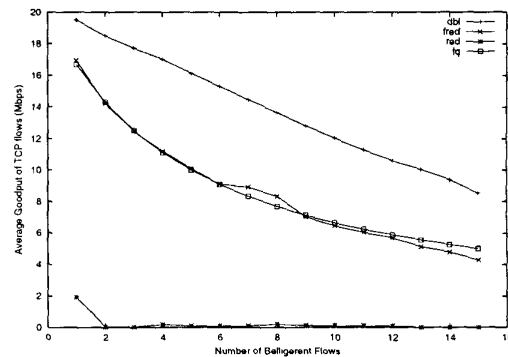


**Figure 5. DBL contains Belligerent Flows providing TCP flows high goodput**

Figure 5 shows the average goodput of TCP flows with DBL, RED, FRED and FQ with increasing number of belligerent flows, from 1 to 15. Again, DBL limits all belligerent flows severely. TCP flows get the highest goodput with DBL as compared to RED, FRED or FQ. With RED, with 2 or more wire-speed belligerent flows, the TCP flows are effectively stopped.

**Table 3. Average Goodput (Mbps) per TCP with 200 Belligerent Flows**

| | tcp-0 | tcp-1 | tcp-2 | tcp-3 | tcp-4 | per bf |
|---|---|---|---|---|---|---|
| DBL | 1.9 | 1.5 | 1.6 | 2.4 | 1.6 | 0.47 |
| FRED | 0.4 | 0.4 | 0.5 | 0.38 | 0.44 | 0.49 |
| RED | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | 0.49 |
| FQ | 0.49 | 0.49 | 0.49 | 0.49 | 0.49 | 0.49 |

Finally, Table 3 summarizes the results with *many* belligerent flows, namely 200. Again, DBL contains all the

belligerent flows, and TCPs get the best goodput. DBL classifies all 200 belligerent flows as such, and TCP flows only enounter single probabilistic drops with no misclassification. With RED, the distribution of the bandwidth among the belligerent flows had high variance, where many of the flows were completely denied and others received high bandwidth.

### 4.4. Multiple Belligerent Flows

Figure 6 shows the buffer consumption of a set of 10 belligerent flows over time, each coming from its own 200 Mbps link destined to the same 100 Mbps output link. Each belligerent flow is pulled down to `bfBufferLimit` cells after recognition as a belligerent flow. With increasing buffer usage, `dbl` first goes down to `dblMin`. After belligerent flows are classified as such, their incoming packets are dropped and the existing buffers in the queue are drained. With the decreasing buffer usage in the queue, `dbl` increases back to `dblMax` while belligerent flows are kept at `bfBufferLimit`.

Compared to the single 200 Mbps belligerent flow classification in Figure 2, the belligerent flow identification time for 10 200 Mbps flows is shorter because of higher congestion in the output queue.

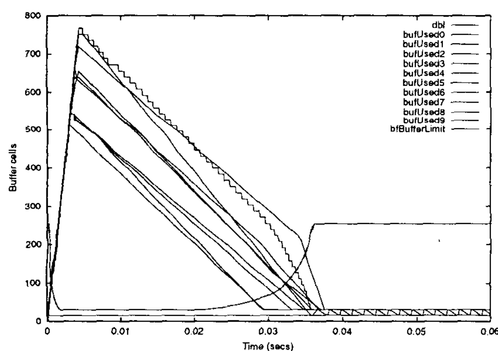


**Figure 6. DBL limits buffering given to each belligerent flow after recognizing them. The bottom straight line at 16 cells is bfBufferLimit. The curved line that goes down to 32 cells first and then increases to 256 cells is dbl.**

We also simulated with multiple belligerent flows of varying rates such as each belligerent flow arriving with a random rate between 100 and 200 Mbps. DBL successfully recognized all belligerent flows and kept them under `bfBufferLimit`.

### 4.5. Multiple Congested Hops with Belligerent Flows

This simulation shows how DBL behaves with multiple congested hops and belligerent flows. Figure 7 illustrates the network configuration we used. `TCP src` node acts as an FTP server with the client on the `sink` node. Each `bf` node sources 100 Mbps wire-speed traffic destined to `sink`. Thus the total number of active flows on the last congested hop, `hop n`, is equal to $n + 1$.

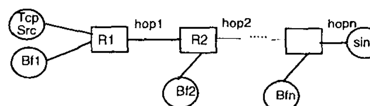


**Figure 7. Multi-hop Scenario**

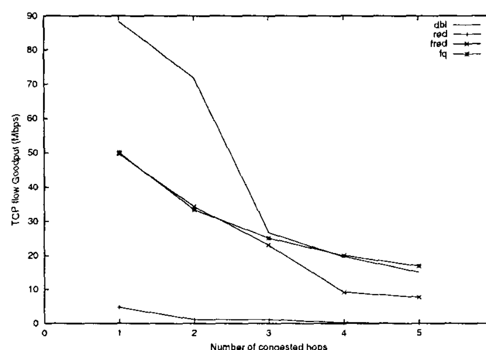


**Figure 8. TCP Goodput under multiple congested hops with belligerent flows**

Figure 8 shows the goodput obtained by the TCP flow with DBL, RED, FRED and FQ with increasing number of congested links. The single TCP client on `sink` obtains the higher goodput with DBL compared to RED and FRED because belligerent flows are severely limited. With RED, the TCP flow achieves very low goodput, effectively stopped. Compared to FQ, DBL achieves better or comparable performance.

These simulations are a small sample of possible network situations, but do show that DBL works as designed. These results support our contention that DBL identifies and constrains belligerent flows quickly and simply.

## 5. Handling Large Numbers of Belligerent Flows

DBL is designed to support a large number of flows, belligerent and otherwise. DBL has very low state cost per ac-

tive flow as discussed in Section 3. This allows supporting a large flow table so that the switch first hits other resource limits as the number of flows increases. If a queue can at most accomodate $n$ packets (which also means it can accomodate at most $n$ active flows), then its DBL flow table can be configured as $4 * n$ or $8 * n$ entries allowing for hash inefficiency to make hash collisions low probability events. Then with increasing number of flows the queue space is the limiting factor, not the flow table.

The other potential limit arises from *significant* hash collisions between flows. A collision of a fragile or robust flow with a belligerent flow is *significant*, because the colliding flow is handled as a belligerent flow. It is limited to lower buffer limits as long as the belligerent flow is active in the switch. We designed DBL so that the hash function into the flow table can be changed during switch operation to prevent persistent collisions.

The probability of a *significant* collision is low because most active flows are non-belligerent and thus the most likely collision candidates. Collisions among non-belligerent flows only require these flows to adapt to a reduced buffer limit. This is due to the sharing of a flow entry and thus have little impact on performance. The small number of belligerent flows relative to the total number of flows in a network reduces the probability of a good flow colliding with a belligerent flow to a very low value. If there are $n$ belligerent flows and the hash table has $b$ buckets, then the probability of a flow colliding with one of the belligerent flows is $1 - [1 - \frac{1}{b}]^n$. Figure 9 shows the probability function when the per-queue flow table has 4096 entries, i.e. $b = 4096$, used for a queue that has space for 512 to 1024 packets at most, i.e. a maximum of 512 to 1024 active flows.

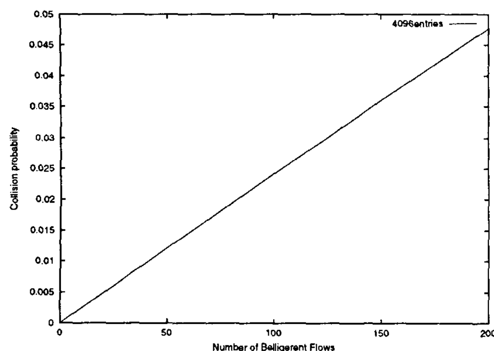


**Figure 9. Probability of colliding with a Belligerent Flow (hash table of 4096 buckets)**

As seen, when $n$ is low, the collision probability is negligible. When the number of belligerent flows increases to 40, the probability of colliding with a belligerent flow is only 0.97%. When the number of belligerent flows increases to 200, namely 20% of total number of active flows, the collision probability increases to 4.7%. Given the queue for this size of flow table is 512 to 1024 packets, with 200 belligerent flows the switch would run out of queue space first.

DBL can be configured with per-queue flow tables or a shared flow table. With the former, there is a flow table per queue providing complete isolation among ports and queues so that congestion on one port cannot affect another port. With a shared flow table, multiple ports can share a bigger portion of the flow table. In this configuration, the flow table hash function takes the output port and queue number as input in addition to the flow label to reduce the likelihood of a collision between flows going to different output ports. This configuration provides a larger flow hash table, thus reducing the probability of hash collisions even further. For instance, with 200 active belligerent flows in a 512K entry hash table, the probability of colliding with a belligerent flow is only 0.03%.

One possible extension to belligerent flow handling is to track the input port of belligerent flows. An input port that is responsible for an excessive number of belligerent flows could be throttled automatically or shut down pending administrator intervention.

If certain type of traffic is not to be dropped as long as buffers are available in the switch, DBL can be turned off for such traffic as specified by the packet classification process. Alternatively, certain traffic can be imposed to hard drop after hitting `dbl` buffer limit instead of probabilistic drop.

A concern with DBL and any other buffering based queuing control mechanism is the dependence of their performance on packet arrival times. Theoretically, one can time belligerent flows such that each belligerent flow adds `dbl` amount of buffering to the output queue, consuming the maximum possible amount of buffering and leaving less to other flows. In reality, this would require a malicious user to know how the dbl parameters are programmed into the switch and also control all the traffic going to that output queue, a very unlikely scenario.

Let us now compare the cost of DBL and its support of many flows to alternative designs such as per-flow queuing and exact match flow mapping.

## 5.1. DBL vs. Per-Flow Queuing

One alternative is per-flow queuing implemented in hardware. However, per-flow queuing supports less than one third of the number of flows as DBL for a given cost or technology-limited amount of memory. We assume that the per-flow queue is selected by a hash from the packet flow label, as in SFQ [10], with the same number of entries as

the DBL flow table. Assuming a 512K entry hash table and 25% utilization, both switches support roughly 128K flows. We further assume the same amount of queue memory for both switches to store necessary packet control information, say an 8 Mbit SRAM with 256K entries, thus addressed by 18 bits. Similar to DBL, this allows a minimum of 2 queue entries per flow queue at worst when all 128K flows are active. With class-based queuing, assuming 64 ports with 4 queues per port, a total of 256K queue entries support 1K entries per queue. Per-flow queues are implemented using a linked list implementation for efficient use of this queue memory by dynamically growing a flow queue. Otherwise a larger queue memory is needed to support the same number of flows because flow queues have to be statically allocated. A 256K x 18 bit link address memory is needed to store the address of the next entry in the queue memory.

Per-flow queuing has $512K * 36bits = 18Mbits$ of head/tail pointer memory, $256K * 18bits = 4.5Mbits$ of linked list space, and $512K * 13 = 6.6Mbits$ to store the queue length (per queue)[7], a total of 29.1Mbits or 227 bits per flow. This is 3.2 times as expensive as DBL. Thus, a competitive product would support 1/3rd of the flows that a DBL-based switch could support for the same cost.

Per-flow queuing is also far more complex to implement. In high speed switches, packet processing cycles are a critical resource. If two cycles are used for reception and two for transmission per packet (allowing a read and write in both directions) in a 125MHz switch with an 8 ns clock cycle, each 16 ns the next packet to be scheduled has to be determined with a central scheduler. Providing more cycles per packet means reduced throughput and thus fewer ports in the switch and higher per port cost. This is undesirable for cost-sensitive enterprise market. With potentially fifty thousand or more flows active in the switch, selecting the next queue to schedule in 16ns is challenging even with today's advanced VLSI technologies. Alternatively, a per-port scheduler can be used, allowing more cycles to select the next queue but resulting in a significantly more logic.

In summary, per-flow queuing is more expensive, producing a mechanism that handles fewer flows for the same cost. This is a significant exposure given that an excessive number of flows is perhaps the most significant concern with per-flow queuing, with DBL, and basically all of the alternative per flow schemes.

### 5.2. Exact Match Flow Mapping

Another design alternative to DBL is to use an exact match flow mapping to detect and handle hash collisions. However, this approach requires storing the flow label per entry. This is 7 times bigger than a DBL entry for IPv4 using 121 bits vs. 17 bits. Thus, an exact match approach would only support roughly 18K flows with the same amount of memory as DBL uses to support 128K flows. This leads to hash collisions at much lower load. Moreover, it is then unclear how to deal with hash collisions. It is not practical to handle the packets in software at the speeds of interest, and just dropping the packets can lead to worse behavior than from a hash collision in DBL.

[7]The queue limit can be just one per class so is negligible space.

## 6. Related Work

Flow RED [8] describes a belligerent flow mechanism similar to ours in using "strikes" versus our credits. However, just like RED, it relies on computing the average queue length, requiring a complex computation that is challenging to perform at very high speed in hardware due to floating point operations and exponential averaging. Moreover, FRED requires this average to be updated on enqueue as well as dequeue, requiring the hardware to be duplicated or else twice as fast as if it was just performed once. In contrast, the DBL computation is completely programmable by using a table lookup that is only accessed on input, enabling a simpler and faster logic realization and allowing local network policies to be implemented. Thus, DBL appears more practical for hardware implementation, especially since our simulations indicate that DBL performs comparably to Flow RED (with the more realistic switch parameter of 1000's of packet buffers versus the FRED simulations with 10's of packet buffers).

BLUE [5] and specifically stochastic fair BLUE (SFB) also attacks the issues of belligerent flows. BLUE uses multiple hash tables to reduce the probability of a belligerent flow collision with another flow. However, this approach requires multiple parallel updates per packet (one per hash) and a large size of `bufferUsed` field, because each entry is typically updated by multiple flows. The additional costs incurred with BLUE are the extra memory chips and the extra pins on the switch ASICs to support multiple memory accesses and wider memory fields. In contrast, we exploit having a large number of entries to minimize collisions and use a single read-modify-write access to the memory. Furthermore, their results are based on 23 and 46 buckets, which are not indicative of real switch parameters when a single low-cost memory chip can provide 512K entries. DBL is simpler and cheaper to implement to support a given number of flows and is implementable in high speed hardware.

Guerin et al. [7] describes the centrality of buffer management for quality-of-service including handling belligerent flows, recognizing as we do that buffer management is simpler and more essential than sophisticated output queuing schemes. However, their focus is on controlling flow rates according to a policy. They do not identify a mecha-

nism for buffer management or provide results on controlling belligerent flows.

Per-flow queuing as in WFQ [3] provides flow isolation like DBL, but DBL requires less memory and is thus cheaper to implement. DBL is also easier to implement in high-speed switches, and WFQ without buffer control does not protect against belligerent flows.

As another category of techniques, RSVP [2] used on a per-flow basis (and similar explicit resource reservation schemes) is impractical to apply to *all* flows given the dominance of best-effort datagram traffic. This leaves the issue of handling these datagram flows, including belligerent flows, open.

Most other work has been focused on providing "fair" sharing between competing, but well-behaved flows and assumes that buffering is not a critical problem. For instance, RED [6, 12] and its variants have been shown to provide fairness (and good throughput) among well-behaved TCP flows but lack any practical realization of the "penalty box" mechanism for dealing with belligerent flows, especially in high-speed environments. Without a belligerent flow mechanism, a switch running RED or its variants can run out of buffering, offering no protection of QoS.

Recent work by Floyd [9] explores extensions to RED to provide the penalty box mechanism. However, this mechanism appears more oriented towards software routers, being quite complex to implement in high-speed hardware because it requires dynamically compiling multiple flow drop history lists in addition to monitoring identified flows in a hash table or similar structure. Moreover, it appears to take much longer to react to misbehaving flows and it does not protect fragile flows.

Our focus with DBL has been on high-speed switches. Low-speed devices can use inexpensive DRAM and a software implementation of WFQ or other mechanisms, so DBL is less compelling there. Adapting DBL for wide-area routers, one can generally afford to put a large packet buffer memory on output line cards because of the expense of the WAN links. In this case, it is feasible to use a large amount of memory and use DBL with a network policy tuned to WAN roundtrip times and acceptable client data rate limits.

Much of this prior work has assumed relatively low data rates, short transmit queues and very limited buffering (10-30 packets per port). With the falling cost of high-speed memory, and transmission rates increasing to 10 Gbps, these characteristics have fundamentally changed in the enterprise environment, with other types of networks showing prospects of following this trend.

## 7. Conclusions

Dynamic Buffer Limiting (DBL) provides an effective protection mechanism against belligerent flows. It protects against a switch being forced into indiscriminant packet drop because of packet buffer exhaustion, reacting and restricting belligerent flows within the time constraints of the buffering in high-speed switches.

DBL has been implemented at wirespeed on all ports in high-speed switch hardware at low-cost. It has the lowest cost per flow of any of the schemes we have considered. The more efficient use of memory means that DBL can handle a larger number of flows for a given cost than it could otherwise.

There are several noteworthy aspects to the design of DBL.

First, DBL focuses on buffer consumption as the key issue. This directly protects the most critical resource in the switch (i.e. buffers), indirectly controls the occupancy of the queues, and thereby limits the flow data rates. Managing buffers per flow has far less space overhead than providing per-flow queues yet approximates the benefits of the latter at the high data rates we are focused on. In general, DBL is a design point between RED, which explicitly avoids per-flow state, and WFQ which uses per-flow queues. It uses memory to simplify belligerent flow handling compared to RED's avoidance of per-flow state yet it uses this memory more efficiently than WFQ.

Second, due to low DBL state cost per flow, the switch runs out of other resources with large number of active flows before it runs out of DBL flow entries. The hash table approximated mapping of flow to per-flow state is more effective for a given amount of memory than a directory storing a full flow label.

Third, DBL maintains state on all active flows all the time, not just those of concern. This structure allows DBL to react quickly and accurately to belligerent flows. It also allows DBL to strictly protect fragile flows and signal TCP flows accurately when each approaches its fair buffering limits.

Finally, the simple hardware logic of DBL supports wirespeed operation in high speed switches.

With per-queue management of resources in DBL, different qualities of service are provided to different classes of traffic by using multiple queues per port, one queue per class. Each queue can be allocated a share of the output bandwidth as well as a shaping rate and a priority. With the small number of classes of service, the cost of the queues is far lower than per-flow queuing while still providing a high degree of flow isolation as a result of DBL. Our full report [4] describes QoS with DBL in more detail.

Overall, hardware QoS protection mechanisms can be expected to become more important as networks move to far higher speed and network resources are primarily strained by failures and attacks. DBL is a strong candidate for such a mechanism in the enterprise campus network.

## Acknowledgments

We thank to ICNP reviewers for their feedback. We thank to Hugh Holbrook and Adam Sweeney for their comments on the paper.

## References

[1] B. Braden, D. Clark, J. CrowCroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in internet, April 1998.

[2] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp), September 1997.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM 1989*, pages 3–26. ACM SIGCOMM, 1989.

[4] F. Ertemalp, D. Cheriton, and A. Bechtolsheim. Dynamic buffer limiting and class based queueing: Qos in high-speed networks. ftp://ftpeng.cisco.com/ertemalp/dbl/, 2001.

[5] W. Feng, D. D. Kandlur, and K. S. D. Saha. Blue: A new class of active queue management algorithms. Technical report, MIT, April 1999. Tech Report available from http://www.eecs.umich.edu/ wuchang/blue.

[6] S. Floyd and V. Jacobson. Random early detection for congestion avoidance. *IEEE/ACM Transactions on Networking*, Aug. 1993.

[7] R. Guerin, S. Kamat, V. Peris, and R. Rajan. Scalable qos provision through buffer management. In *Proceedings of SIGCOMM 1998*, pages 29–40, Vancouver, BC, Canada, September 1998. ACM SIGCOMM.

[8] D. Lin and R. Morris. Dynamics of random early detection. Revised version of ACM SIGCOMM 1997 paper.

[9] R. Mahajan and S. Floyd. Controlling high bandwidth flows at the congested router. In *Proceedings ICNP 2001*, Riverside, Ca, November 2001.

[10] P. McKenney. Stochastic fairness queueing. In *Proceedings of the Conference on Computer Communications*. IEEE infocom, 1990.

[11] Lbnl network simulator, version 2. http://www-mash.cs.berkeley.edu/ns/ns.html.

[12] B. Reynolds. Red analysis for congested network core and customer egress. http://null0.qual.net/brad/papers/reddraft.html.

[13] I. Stoica. ns-2 fred simulation from csfq work. *http://www.cs.cmu.edu/ istoica/csfq*, 1998.