Quality of Service Provisioning for Composable Routing Elements

Seung Chul Han Puneet Zaroo David K. Y. Yau Yu Dong

Prem Gopalan John C. S. Lui*

Abstract

Quality of service (QoS) provisioning for *dynamically composable software elements* in a programmable router has not received much attention. We present a router platform that supports extensible and configurable routing elements, and provides them with access to given resource allocations. Scheduling issues for these elements are discussed: (1) flow-based scheduling, (2) the preemptibility of a pipeline of elements, (3) CPU conservation for idle elements, (4) the CPU balance between input, output, and processing elements and its effects on buffer provisioning, and (5) performance interactions between the packet forwarding plane and the service extension control plane.

*P. Zaroo is with Vmware, Palo Alto, CA; P. Gopalan is with Mazu Networks, Cambridge, MA (work done while S. C. Han, P. Zaroo, and P. Gopalan are graduate students at Purdue University); D. K. Y. Yau and Y. Dong are with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907; J. C. S. Lui is with the Department of Computer Science and Engineering, the Chinese University of Hong Kong.
Research was supported in part by the National Science Foundation under grant numbers CCR-9875742 (CAREER) and CNS-0305496, by a Hong Kong RGC earmarked grant, and by an IBM Ph.D. Fellowship awarded to Y. Dong.

To demonstrate how QoS provisioning in our system can benefit end users, we use a video scaling application that can respond gracefully to network congestion. For the application, we quantify how router resource management impacts the end-to-end quality of decoded video. Ours appears to be the first comprehensive experimental evaluation of a software system that supports QoS-aware processing of lightweight, dynamic router elements.

1 Introduction

Value-added processing of packets during their transport, especially at the network edge, is increasingly relevant. Example applications include security firewalls, network address translations, and proxy services to adapt application payload (e.g., a movie being streamed) to network conditions. Moreover, some of these services are not anticipated in advance. For example, in response to emerging security threats, new defense mechanisms will be designed as countermeasures. (Previous instances include proposals such as IP traceback [9] and route-based packet filtering [7] to defend against distributed denial-of-service attacks.) Hence, the ability to extend the service interface of a router or proxy server on the fly, without disrupting existing services, is attractive.

In providing extensible, value-added services during packet transport, we adopt an approach based on *software elements*. An element is a self-contained code module implementing a logical routing function. The advantages of using these routing elements are many:

- The elements can be composed to form a flow processing pipeline. Hence, more complex router services can be constructed from simpler and well understood building blocks. This has important software engineering benefits, by isolating design

and implementation concerns and facilitating code reuse.

- An element implementing a common routing function can be shared by several flows desiring the function. This contributes to code and memory efficiency.

- Elements can be easily mapped to a lightweight execution context. For example, different elements, possibly belonging to different flows, can be executed in the context of a single thread or process. The overhead of context switching between elements or flows can thus be minimized for higher efficiency.

- Elements can be fetched on demand from a (possibly remote) service repository, and dynamically linked into the runtime environment of a router. This enables the service interface of a router to be extensible on the fly, without disrupting existing flows. Services that are hitherto unanticipated can thus be readily introduced into an operational routing infrastructure.

While routing elements have been advanced in prior research and are supported in existing systems (e.g., [6]), their scheduling issues for providing quality of service (QoS) to network flows have not received much attention. In this paper, we present the CROSS/Linux router platform that supports configurable flow graphs of router elements as provided by the Click modular router [6]. Our research contributions beyond Click are in the area of element-related resource allocation and scheduling, which includes the following issues:

- The provision of flow-based resource allocation and scheduling on top of an element-based software architecture.

- The preemption granularity of flow processing. Our system can context switch (with acceptable overhead) from a lower priority flow to a higher priority flow in

the middle of processing a packet. This reduces the duration of priority inversion. We study the resulting effects on robust forwarding of network flows with fine time-scale QoS requirements.

- CPU conservation for "idle" elements (i.e., elements which need not run because their packet queues are empty). We provide an architecture in which elements do not have to poll for work to do.

- The CPU balance between the element functions of input, output, and per-flow processing. We study how giving different CPU shares to these functions will affect buffer provisioning and packet forwarding performance.

- The provision of a service control plance, and accompanying resource contention issues between the forwarding and control planes. In particular, we discuss how the concurrent tasks of flow processing and service downloading may affect each other's performance.

Our work addresses the problem of QoS provisioning at a single router. For end-to-end QoS provisioning over multiple hops, an RSVP like protocol generalized to reserve CPU time will have to be deployed. Although such RSVP style support is not evaluated in this paper, it can naturally run on the control plane of our router.

1.1 Paper organization

The balance of the paper is organized as follows. In Section 2, we review the Click modular router architecture, which provides background for configurable elements being used in our system. We then go on to discuss the design and implementation of CROSS/Linux.

Section 3 presents the forwarding plane for packet processing. Issues for per-flow resource scheduling will be discussed. Section 4 presents the control plane. In particular, it describes the processes of flow signaling and on the fly service extension. CROSS/Linux has been implemented on a network of commodity Pentium III desktops configured as gateway routers. We present measurement results on various aspects of QoS provisioning in our system prototype. Related work is discussed in Section 7. Section 8 concludes.

2 Background

The starting point of our work is the existence of an element-based router architecture, such as provided by the Click modular router [5], in which elements can be configured for customized per-flow processing of packets. For completeness, we briefly review the Click software architecture. In Click, elements are C++ kernel modules each implementing a simple router function (e.g., receive from an input network interface, send to an output interface, packet classification, queuing, and packet scheduling). Elements can be considered nodes in a directed graph, and they can be connected to each other through one or more *ports* they have. When an output port of an element is connected to an input port of another element, it forms a directed edge from the former (the *upstream* element) to the latter (the *downstream* element). A packet can then be passed from the upstream to the downstream element. In general, a packet arriving at an input interface of a router is first processed by an *input* element, where the packet gets classified to its flow. According to the classification, the packet then flows along the edges of the flow graph, from an output port of each upstream element to an input port of each downstream element. It will receive customized protocol processing according to the actual path it traverses, and finally gets forwarded out of the router by an *output* element.

An upstream element initiates packet transfer to its immediate downstream neighbor by calling the *push* virtual function of the neighbor. Hence, packet transfers initiated from upstream (e.g., by network input) are called *push processing*. It is also possible for a downstream element to request packets from upstream (e.g., when an output network interface becomes ready, it may request a packet to send). This is done by the downstream element calling the *pull* virtual function of its immediate upstream neighbor. Hence, packet transfers initiated from downstream is called *pull processing*. Conceptually, push/pull processing is enabled by the arrival of packets at relevant packet queues, and a packet queue in Click is represented by a *Queue* element.

Fig. 1 illustrates a sample flow graph implementing a traffic conditioning block. The graph has two Queue elements – one upstream of the Shaper element and the other downstream of the Meter element. In the example, push processing starting at the Classifier element is enabled by packet arrivals at the input device queue (not shown) served by the classifier, and pull processing starting at the DeviceOutput element is enabled by packet arrivals at either of the two Queue elements shown.

Click has to schedule the execution order of *eligible* elements. Our definition of an eligible element is one that is the starting point of push/pull processing and has available packets to process in the relevant packet queue(s). From the scheduling point of view, a sequence of push (or pull) function calls cannot be interrupted. A packet must pass through the corresponding sequence of elements, until it is either dropped, or queued in the context of a Queue element. For example, the Classifier-Meter-Discard element sequence in Fig. 1 cannot be preempted in the middle. After a packet is dropped or queued, however, the element scheduler regains control, and schedules a next element to run. Hence, the position of Queue elements in a processing path determines the path's

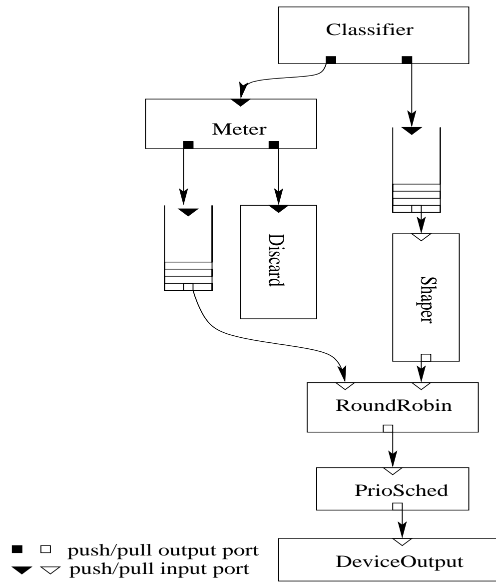

Figure 1: A sample Click flow graph of elements.

preemption granularity in Click scheduling. If more elements are connected in tandem without interposing Queue elements, the preemption granularity becomes coarser, since the scheduler must wait for all the elements to complete before it can reschedule.

3 Forwarding Plane Packet Processing

A fundamental design decision about CROSS/Linux is the scheduling paradigm that should be used for packet processing. A simple approach would be to schedule elements as independent entities, without reference to their execution context. Click chooses such an approach. However, packets sent through a router usually belong to higher level logical flows, which have their own QoS constraints. For example, a video flow may need some minimum forwarding rate to achieve continuity of the pictures. An interactive audio flow may specify some maximum delay bound for its packets, to support high quality voice communication.

To effectively support application-level QoS, we decided to provide a flow abstraction for scheduling the packet forwarding plane. Packets are classified to their flows by a packet classifier, according to *flow specifications* that are installed. For example, a layer-four IP flow can be defined by the source IP address, destination IP address, transport protocol, transport source port, and transport destination port. Router resources can then be allocated on a per-flow basis. In our current model, flows can be given proportional CPU shares or guaranteed CPU *rates*, as provided by start-time fair queueing (SFQ) [2]. Rate fluctuations over the small time scale (e.g., variations in processing time of individual packets) can be handled by either over-provisioning at the peak rate or, more typically, by provisioning at the average rate and using a small buffer to absorb the resulting burstiness. Longer time-scale rate fluctuations can be solved by various rate adaptation approaches (e.g., [14]), which are beyond the scope of this paper. (Other forms of performance guarantees can be provided by other scheduling algorithms. For example, hierarchical fair service curve (HFSC) scheduling [11] can provide explicit delay guarantees.) Hence, as a packet gets processed by the sequence of elements that it goes through, the CPU cycles consumed by the processing are charged to the packet's flow, and not to the elements themselves. In particular, an element being shared by two or more flows consumes resources of the flow being processed. Such decoupling of the resource context from the processing entity is the key to providing performance isolation between logically independent flows.

The CROSS/Linux forwarding plane scheduler (henceforth called the *flow scheduler*) selects the next flow to run from a task queue of all the *eligible* flows in a router. A flow is eligible if one or more of its elements are eligible. Such a flow is represented on the task queue by an *fRouter* abstraction that contains all the pertinent scheduling state about the

flow. Once a flow is scheduled, it still remains to determine the execution order of the flow's eligible elements. We support this next-level scheduling decision by (1) allowing a flow to in turn apportion its CPU allocation among the constituent elements, and (2) maintaining flow-specific scheduling state for each element.

Notice that certain elements do not logically belong to any particular flow. Instead, they perform functions in the *global* router context. Input and output elements for network interfaces, and an element for vanilla IP forwarding, are important examples. We treat these global elements as belonging to certain "global flows". A global flow is represented in the task queue by an *ioRouter* object, a counterpart of the fRouter object for non-global flows. For the purpose of scheduling, global flows are quite similar to normal flows. They can be given specified resource allocations, thus allowing their elements to compete for system resources with other per-flow elements. The assignment of global router functions to global flows is flexible. For example, we could have one global flow for each network input element, one global flow for each network output element, and one global flow for vanilla IP forwarding. Or we could have one global flow for all of network input, network output, and vanilla IP forwarding. Fig. 2 shows a router configuration in which a single ioRouter is used for the router global functions, and two fRouter's have been created for per-flow user processing.

3.1 Preemption granularity

Since a flow represents a line of concurrency, it is natural to run each flow as a separate thread or process. The approach, however, requires high context switching overhead (i.e., one full thread context switch) between flows. To reduce the overhead, previous work [8] has advanced the technique of *batching*, which always tries to process a batch of at least n

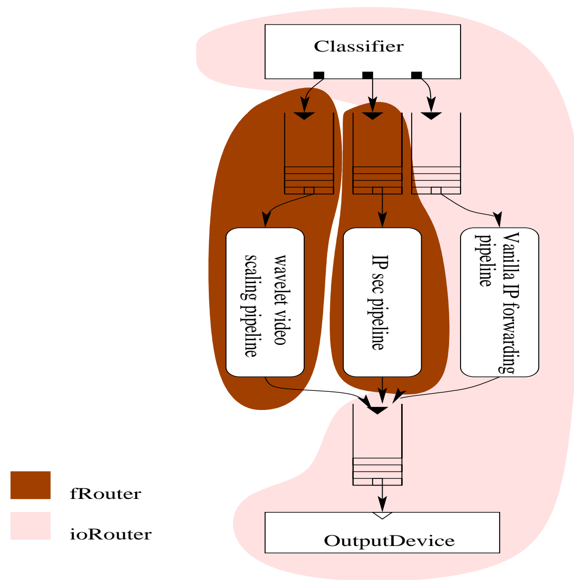

Figure 2: A sample CROSS/Linux router configuration.

packets (provided that these packets are available) belonging to one flow before the system will consider switching to another flow. While batching reduces context switching, it also makes the preemption granularity coarse and hence increases the possible duration of priority inversion. For example, a newly backlogged higher priority flow may have to wait for an entire batch of n lower priority packets to finish before it will get a chance to run.

We have described Click's packet preemption mechanism in Section 2. As discussed, the preemption granularity is a *sequence* of elements that usually ends with a Queue element. This means that a packet can be preempted while being processed. Such smaller preemption granularity than batching is feasible in Click because different packets can be processed by the same thread and no kernel-level thread scheduling is required to switch between them. Since QoS is an important concern in CROSS/Linux and certain applications, like continuous media, may have fine-grained time constraints, we take Click's

approach one step further to allow flow preemption at arbitrary element boundaries.

We associate with each flow, say i, a user-specified *preemption quantum* q_i (in μs) for the flow. The choice of q_i allows users to control the tradeoff between scheduling efficiency and fairness. In general, a smaller q_i gives improved fairness (and hence more precise QoS guarantees) at the expense of a larger context switch overhead. Once scheduled, if i has been running continuously for q_i time, then the system will attempt to reschedule when the current element being processed for i finishes. To do so, before invoking a downstream push call (respectively, upstream pull call) for i's current packet, we check whether q_i has expired or not. If not, we perform the push (respectively, pull) call as usual. If it has expired, however, then instead of performing the push/pull call, the system checks for the need to reschedule. The current packet of i should be preempted if there is another eligible flow in the system that has higher or the same SFQ *virtual time priority* as i. (In SFQ, flows are scheduled in an increasing virtual time priority order [2].) To carry out the preemption, the system saves a pointer to i's current packet and another pointer to the element that should next process the packet when the packet is resumed. Since each element operates on and transforms a packet independently in our system, we do not need to store further execution state for the preempted packet. The added runtime overhead for our preemption mechanism is therefore quite small.

3.2 CPU conservation for idle elements

Recall from Section 2 that, conceptually, flow elements are enabled by packet arrivals into their work queue(s). In practice, however, Click does not distinguish between eligible versus ineligible elements. Instead, elements have to poll their packet queue(s) for work to do. When an element is scheduled but finds no packet to process, it simply returns

but remains eligible for the CPU. Since we assign CPU shares to elements, this imposes a problem. Specifically, an element that has no non-empty work queue will keep on polling, thus wasting CPU time, until it has used up its allocated CPU share. Although we are not able to further elaborate, because of limited space, this causes various anomalies in flow scheduling.

To address the problem, CROSS/Linux maintains a task queue of eligible flows only, where a flow is eligible if at least one of its elements is eligible. When an element finishes processing its last available packet, it will enter the sleep state. When all the elements of a flow sleep, the flow itself enters the sleep state and, therefore, it will be removed from the task queue. Hence, it will not be chosen to run by the flow scheduler. Later, when a packet for the flow arrives, the packet will enable one of the flow's elements, which will have the effect of waking up the flow and putting it back on the task queue.

4 The Control Plane

Whereas the forwarding plane processes packet flows, the control plane of a router runs supporting services such as routing (e.g., OSPF, RIP, and BGP) and signaling (e.g., SIP and RSVP) daemons. In the case of an extensible services router, the ability to download code modules on the fly is important. It allows services that are not planned *a priori* to be deployed as they become available or as the need arises. For this purpose, the DARPA active network project has developed the active network daemon, called anetd, for fetching code from a remote repository. We leverage anetd in providing on-demand service extension. System support for interfacing CROSS/Linux with anetd is discussed in Section 4.1.

Control plane services usually run as user-level processes. Fig. 3 illustrates how such

a service can be started. In the figure, a request to start anetd is received by the router, and causes the anetd daemon process to be spawned. After startup, the daemon "subscribes" to anetd packets through a standard socket-type API. This installs a new rule in the packet classifier for anetd packets to be locally queued for reading by the daemon. Future anetd packets will thus be delivered to the daemon, instead of being forwarded by the router.

Processes in the control plane compete for system resources with each other and with the forwarding plane. To schedule the competing demands, CROSS/Linux implements a *system level* multiresource scheduling architecture based on *resource allocations* [13]. Similar to [13], QoS-aware schedulers for CPU cycles, network bandwidth, disk bandwidth and main memory have been integrated, although the current CPU scheduler supports only proportional shares but not decoupled delay and rate allocations. Notice also that the flow scheduler described in Section 3 can be treated essentially as a system process and hence, can be given a CPU share relative to other processes or threads in the system. The flow scheduler then allocates the received CPU share to the packet flows that it manages.

4.1 Flow Signaling and Service Configuration

So far, we have described flow scheduling assuming that the flows have been already set up. CROSS/Linux also allows flows to be dynamically created and flexibly configured as a pipeline of elements. Such flow management is effected by IP control packets with the *router alert* option being set. Three kinds of control packets are defined: `IC_SETUP` for creating flows, `IC_TEARD` for destroying flows, and `IC_CONFIG` for configuring a flow element. The packet classifier reading from an input interface identifies these control packets and delivers them to a control queue. A system *control thread* processes packets

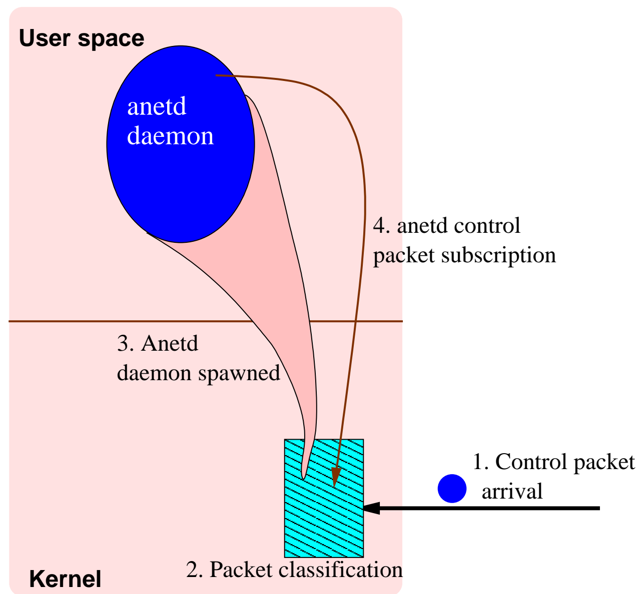

Figure 3: Anetd service startup.

in the control queue in FIFO order. It runs code implemented in a FlowManager element (also called the *flow manager*), which is similar to the original Click element for IP classification, but has additional support for adding new ports and filter rules. Such support is clearly crucial for dynamic flow creation.

4.1.1 Flow setup

When an `IC_SETUP` packet is received, the flow manager constructs a configuration string representing the flow specification encoded in the packet. Once the string is composed, the original set of configuration strings maintained by the flow manager is reconfigured to include the new string. As part of the reconfiguration process, a new element output port is created for the flow manager. The new port is then connected to a Queue element created for the new flow. In addition, an fRouter object will be created and allocated resources according to parameters carried in the `IC_SETUP` packet. Later packets that

match the classification rule for the new flow are then delivered to the corresponding flow queue.

4.1.2 Flow configuration

An IC_CONFIG control packet is used to add/delete an element to/from the processing pipeline of an existing flow. In the case of adding an element, the flow manager checks whether the requested service is already available in a local service repository. If not, it signals anetd to download the named service from a remote node. The anetd daemon looks up the remote node having the service. It then reliably fetches the code, as an uninterpreted byte stream, from a web server running on that node, using HTTP. For CROSS/Linux, the byte stream must correspond to a compiled kernel module for the requesting machine. If the download fails (e.g., the requested service cannot be found) in the current implementation, the request to add an element silently fails, in that the sender of the add request is not notified of the failure. If the download succeeds, the fetched code will be entered into the local service repository. Once the code is available locally, it is dynamically linked with the running kernel using the standard Linux `insmod` utility. Lastly, the linked module is configured into the processing pipeline through the standard Click mechanism of writing a *service specification* to the kernel through the `/proc` file system.

Fig. 4 illustrates the flow configuration process. In the figure, step 2 for spawning a new control thread is optional. In the current implementation, it is invoked only if the control thread is not already running when the IC_CONFIG packet is received. Notice also that code downloading can take place concurrently with normal packet forwarding, and that the code packets returned from the HTTP server are not forwarded but are delivered

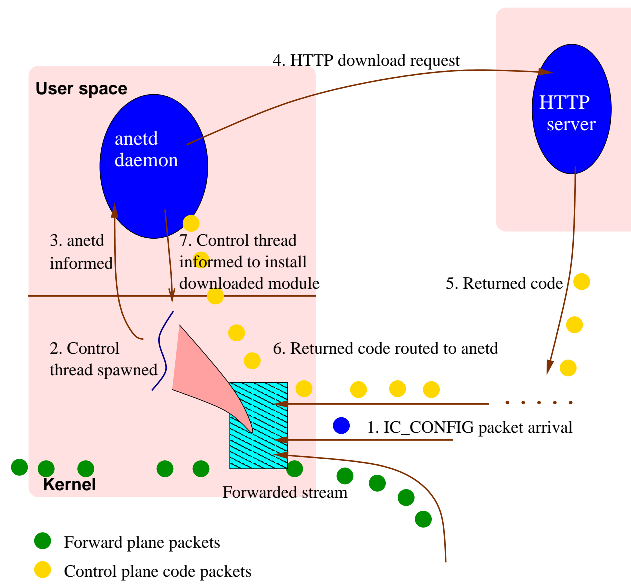

Figure 4: The process of service configuration using anetd.

to anetd. This is because anetd has previously subscribed to the packets.

4.1.3 Flow delete

When an IC_TEARD is received, the flow manager verifies the existence of the named fRouter. If it exists, it is removed from the flow scheduler, its flow specification is removed from the packet classifier, and any memory allocated to it is returned to the kernel.

5 Video Scaling Application

A media scaling service is reported in [4] for router plugins [1]. The service applies to wavelet-encoded real-time video consisting of a base layer and progressive enhancement layers. Lower layers contain more basic video information, and are needed for higher layers to add to the video quality. By using a plugin to examine the layer information of backlogged video packets at times of network congestion, the router can drop enhancement

layer packets before base layer packets, and higher enhancement layer packets before lower enhancement layer packets. This way, it is possible to achieve *graceful degradation* of video quality under constrained network bandwidth.

We have ported wavelet video scaling to CROSS/Linux. The service can be fetched and loaded on demand, in response to user requests. While the same service has been demonstrated in [4], our goal is to understand how resource management in CROSS/Linux can impact video quality perceived by end users. In particular, video scaling requires sufficient CPU cycles to be effective. Otherwise, video packets will be dropped in an *undifferentiated* manner while awaiting processing by the scaling module. We are interested in experimentally assessing how different CPU allocations for the scaling service can affect video quality. Resource allocation issues are particularly relevant for applications like video streaming that have QoS constraints.

6 Experimental Results

We present experimental results to illustrate application performance on CROSS/Linux. The routing platform used is a Pentium III/866 MHz PC fitted with four PCI 3Com 3c59x (vortex) 10/100 Mb/s ethernet interfaces. The original vortex driver runs in *interrupt* mode, in which every packet arrival from the network generates a device interrupt. We have made our own changes to the vortex device driver to additionally support *polling* I/O, in which the device driver polls the network interface for packet arrivals (i.e., there is no interrupt overhead for receiving packets). Polling is much less expensive than interrupt processing, and can significantly increase the efficiency and stability of a router having to deal with frequent packet arrivals [6, 8]. For the global router functions, we schedule them in the context of a *single* global flow, similar to the configuration shown in

Fig. 2. Also, the preemption quantum is set to be $q_i = 5\mu s$. Such a small q_i value effectively causes preemption to be considered after the processing of *every* router element.

6.1 Context switching

As discussed, an element-based architecture allows low context switching overhead between flows, if the flow elements are run in the context of one kernel thread. To verify the claim, we measure the overhead of flow context switching in CROSS/Linux, as a function of the number of eligible flows in the system. Each flow is given the same CPU share and is always enabled. Fig. 5 shows the results. The overhead has two components. First, it has a fixed component of about 280 ns, which includes the tasks of dequeuing the incoming flow from the head of the task list, storing the execution state of the flow being switched out (e.g., the next element to process the flow's packet that is being preempted), and updating the proportional-share scheduling state of both the incoming and outgoing flows. Second, it has a linear component that has a measured value of around 5 ns/flow, which accounts for the time required to insert the outgoing flow into the task list in sorted order of the eligible flows' virtual time priorities. The linear time reflects our current implementation of the task list as a doubly linked list of the eligible flows. A priority queue implementation can reduce the implementation complexity to O($\log n$), where n is the number of eligible flows in the system. To put our numbers in perspective, the reported cost for context switching between forwarding processes in [8] is 3.3 μs, after aggressive performance optimization using continuations.

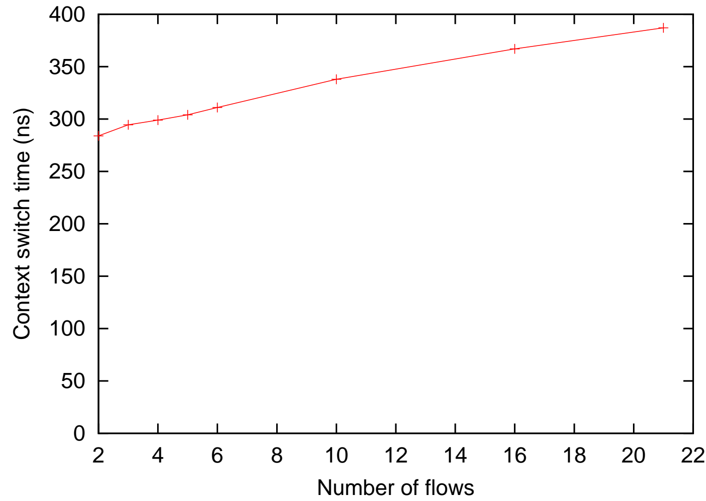

Figure 5: Context switch overhead as a function of the number of eligible flows.

6.2 Throughput comparison with Click

CROSS/Linux has added support for QoS beyond Click, and our measurement platform uses fine-grained preemption at the boundary of every element. We verify that the extra mechanisms do not compromise the system's efficiency in forwarding packets. To do so, we compare the achievable throughput by Click and CROSS/Linux in forwarding small size (specifically, 64-byte) packets, because small packets maximize the demand on CPU processing. We configure ten flows each with equal CPU share. Each flow sends evenly paced packets at a same specified input rate. We vary the aggregate input packet rate from 10K to 90K packets/s for both the polling and interrupt modes. Each flow The results are shown in Fig. 6. For polling, both Click and CROSS/Linux achieve a forwarding rate equal to the input rate (i.e., there is no packet loss) at all the offered loads. For interrupt mode, both Click and CROSS/Linux achieve lossless forwarding at up to about 60K packets/s. When the input rate is 70K to 90K packets/s, losses occur for both systems, and the achieved forwarding rate of CROSS/Linux is about 90% of

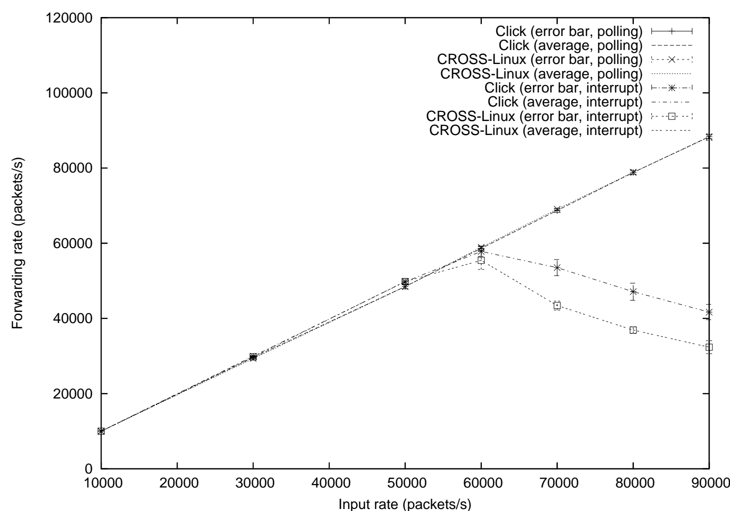

Figure 6: Click and CROSS/Linux packet forwarding performance in polling and interrupt modes.

Click's forwarding rate. We conclude that QoS support in CROSS/Linux does not cause significant loss in system performance, especially in polling mode.

6.3 Forwarding/control plane contention

We examine system performance when the control plane contends with the forwarding plane for resources. To do so, we let our router forward flows as usual. Then, *while the forwarding is going on*, we send an `IC_CONFIG` control packet to download and configure the WaveScaleCOLOR.o module into the running kernel. The system level scheduler in Section 4 is used to allocate relative CPU shares to the flow scheduler, anetd and the control thread that interacts with anetd. In the experiment, we simply use the default scheduling parameters such that the three threads all have the same CPU share. The forwarding plane has much higher actual load than the other two threads, but it can make use of the CPU cycles not claimed by them. No reservation for network bandwidth is made in the experiment.

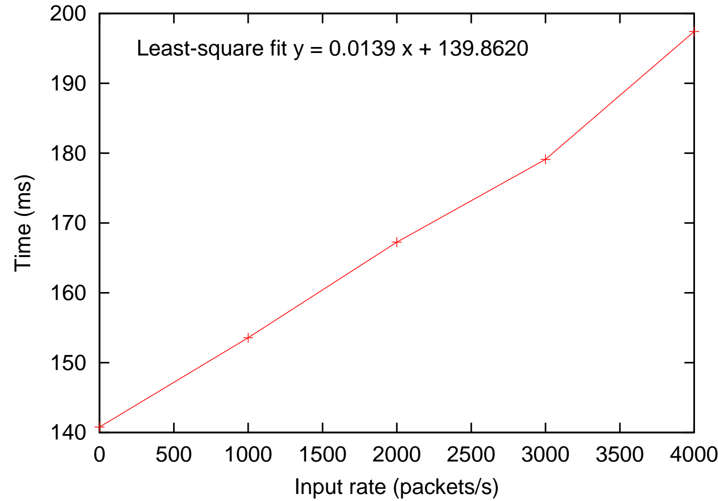

Figure 7: Time to configure WaveScaleCOLOR.o as a function of the competing forwarding plane packet rate.

We vary the offered traffic rate for the forwarding plane from 10K to 100K 64-byte packets/s. We measure the actual forwarding rate achieved by the forwarding plane and also the time taken for WaveScaleCOLOR.o to be successfully installed. From Fig. 7, notice that the configuration time is partly constant and partly linear with the offered traffic rate. Let y (in ms) be the configuration time and x (in packets/s) be the offered traffic rate. We found that a linear least square polynomial, $y = 0.0139x + 139.86$, provides a very good fit with an R-coefficient of 0.9972.

For the achieved forwarding rate, we compare the cases when forwarding occurs with and without competition from the service configuration process. From Fig. 8, notice that there is no observable performance difference between the two cases. We conclude that service configuration requires only a small fraction of the system resources such that it makes no significant impact on the forwarding plane.

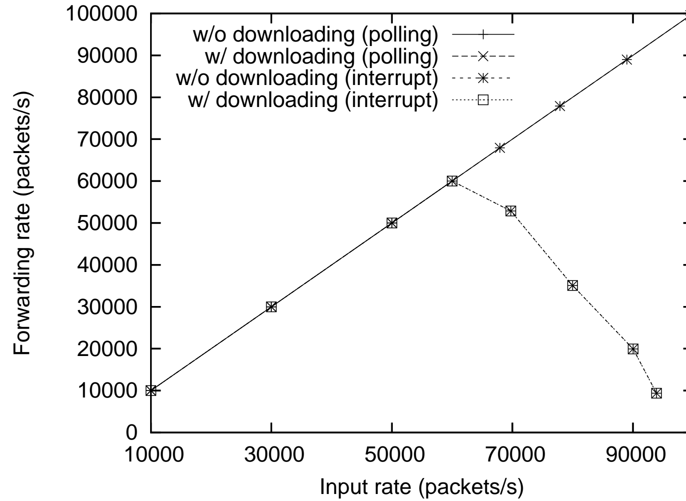

Figure 8: Packet forwarding performance, with and without competing service configuration.

6.4 Flow-based versus element-based scheduling

A fundamental design decision about CROSS/Linux is to impose a flow abstraction over Click's element-based architecture. We demonstrate the performance impact of flow-based scheduling. We configure two flows, A and B, as shown in Fig. 9. Notice that the MultPull2Push element is being shared by the two flows. Our objective is to process flow A with twice the actual CPU capacity as B. In the case of Click, CPU shares are assigned per element. Given the sharing objective, we assign B's private Paint element a CPU share of 2, and A's private Paint element a share of 4. It is not easy to assign a CPU share to the MultPull2Push element, since it is being shared. We make the apparently reasonable choice of assigning it a share of $(2 + 4)/2 = 3$. For CROSS/Linux, CPU shares are assigned per-flow. Hence, we simply assign shares to flows A and B in the ratio of 2:1.

We then generate 64-byte packet arrivals for the two flows so that they are always backlogged. Fig. 10 shows the cumulative CPU consumption of A and B in Click, as a

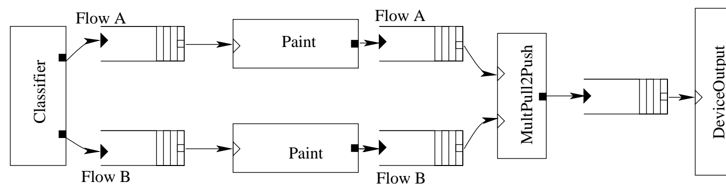

Figure 9: Configuration of Flow A and Flow B to evaluate flow-based versus element-based scheduling. Notice that MultPull2Push is shared by both flows.

function of time. Given the progress rate of B, the *expected* progress rate of A is also shown for comparison. Notice from the figure that the actual rate of A is significantly smaller than the expected rate. This is because the MultPull2Push element does not get sufficient CPU cycles to keep up with A's packet arrivals, causing the packets to be dropped. On the other hand, increasing the CPU share of MultPull2Push gives B the potential to be overly aggressive and take away A's intended share. The result demonstrates the difficulty of assigning appropriate CPU shares to shared elements in Click, such that the logical flows will get their desired actual CPU shares. In contrast, Fig. 11 shows the progress rates of the two flows in CROSS/Linux. Notice that our straightforward flow rate assignments easily result in the desired progress ratio of 2:1 for A relative to B. We conclude that flow-based scheduling avoids complex rate assignment problems when elements can be shared between flows. It thus enables simple and intuitive user control over system resource allocations.

6.5 CPU and buffer provisioning

Packet arrivals from the network may happen quickly relative to the scheduling of the software that processes the packets. If the software cannot run as soon as the packets arrive, the packets may be lost unless there are sufficient buffers to absorb the burstiness.

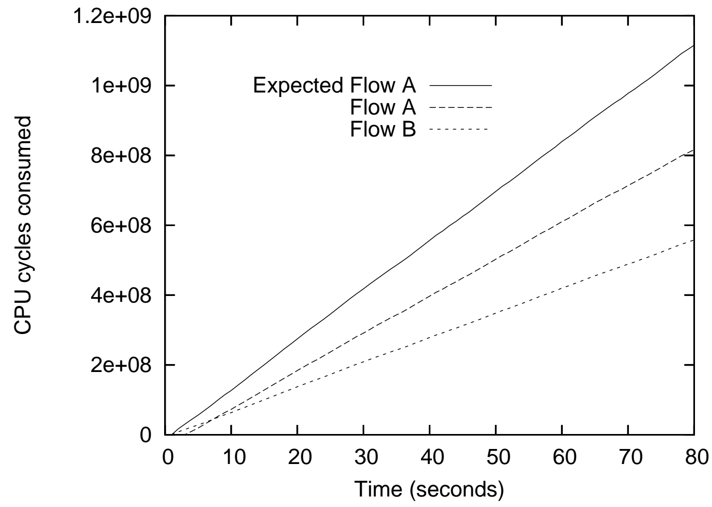

Figure 10: Progress of Flow A and Flow B under element-based scheduling. Notice that A's progress rate deviates from the expected rate.

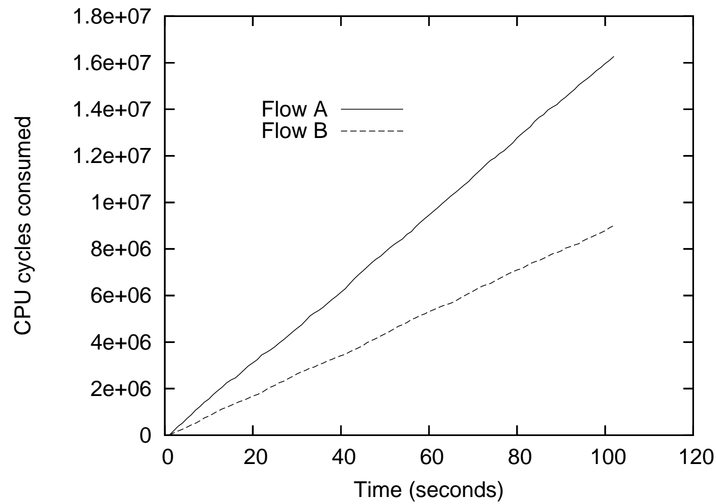

Figure 11: Progress of Flow A and B under flow-based scheduling. The progress rate A to B is very close to the expected ratio of 2:1.

Such loss may occur, for example, at the hardware network interface, if the input element cannot read the packets and classify them quickly enough. It may also occur at a per-flow packet queue if the per-flow element(s) cannot consume the packets fast enough. We examine several issues that affect buffer provisioning in our system to achieve lossless forwarding of packets.

6.5.1 CPU balance

Consider a general flow processing pipeline consisting of three stages: input, per-flow processing, and output. CROSS/Linux can assign different relative CPU shares to the three parts. Let i, f, and o denote the CPU shares given to input, processing, and output, respectively. The ideal ratios between the quantities should depend on the time taken by the corresponding stages. If a function is given too small a CPU share, packet loss may result if the function is not able to keep up with the packet arrivals.

In an experiment, we configure a flow whose input, processing and output stages take about 150 ns, 1.27 μs, and 130 ns, respectively. (Hence, the "ideal" CPU balance between the three stages should be about 1:8:1.) We generate back-to-back 64-byte packets for the flow at a rate of about 30K packets/s. In a set of runs, we allocate CPU shares for input, processing and output in ratios of $1 : f : 1$, where f is varied from 1 to 30. We then measure the *minimum* buffer size (in number of packets) needed for the flow to achieve lossless forwarding of its packets in each run. Polling mode is used. The results for both Click and CROSS/Linux are shown in Fig. 12. Notice that when f is small, a large buffer size is needed in both systems to prevent packet loss (for $f = 1$, Click requires 330 packets and CROSS/Linux requires 310 packets). As f increases, the required buffer size decreases rather quickly, until f reaches about 8 which reflects the ideal CPU balance. In

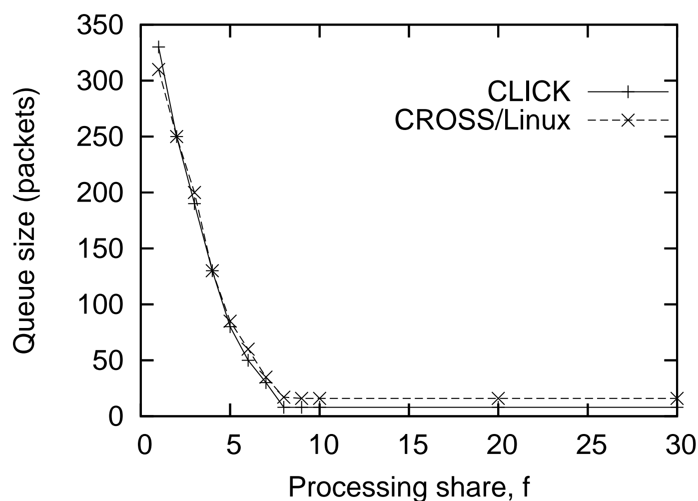

Figure 12: Minimum queue size for lossless forwarding as a function of the processing share f, for both Click and CROSS/Linux.

Click, the required buffer size first reaches the minimum value of 8 packets when $f = 8$ and stays the same when f further increases. In CROSS/Linux, the required buffer size is 17 when $f = 8$ and is 16 when $f = 9$ or higher. The buffer size stabilizes at different values for Click and CROSS/Linux because the two systems have different implementations of the input and queue elements, but further investigation is needed to pinpoint the exact reasons.

In another experiment, we construct another input-processing-output pipeline where processing corresponds to vanilla IP forwarding of packets. We allocate CPU shares to the three stages corresponding to their ideal balance of about 1:10:1. We generate back-to-back 64-byte packets for the flow at a rate of x packets/s, where x is varied to be 9927, 29937, 49355 and 70499 packets/s in a sequence of runs. We then measure the achieved forwarding rate for the flow when the buffer size, denoted by b, is set to be 10, 100, and 1000 packets in different runs.

Table 1 shows the results for polling mode in CROSS/Linux. Notice that when b is

Input rate (packets/s)	Forwarding rate (packets/s)		% forwarded	
	$b = 10$	$b = 100/1000$	$b = 10$	$b = 100/1000$
9927	9887	9927	99.6	100
29937	29833	29937	99.6	100
49355	49144	49355	99.5	100
70499	70198	70499	99.5	100

Table 1: Vanilla IP packet forwarding rate and percentage for buffer sizes of 10, 100 and 1000 packets, and at different offered 64-byte packet rates. Polling mode.

100 or 1000 packets, forwarding is lossless. When b is 10, however, some loss is observed, and the percentage of forwarded packets ranges from about 99.6% to 99.5%. In the case of interrupt mode, the loss rates vary much more for the different buffer sizes. The results are shown in Fig. 13. Notice that for interrupt, a large buffer size (of about 1000 packets) is needed to realize the packet forwarding capacity of the router.

6.5.2 Preemption granularity

The preemption granularity of the system, as discussed in Section 3.1, will also affect buffer provisioning to achieve lossless forwarding. This is because when the preemption granularity is coarse, then a flow (even if it has a sufficient long-term CPU rate to process its packets) may have to wait longer before it will be given a chance to run. If packets arrive for the flow during this waiting period, they will have to be buffered. Then, when the flow runs, it may process a large number of backlogged packets in a burst. Hence, processing for the flow may appear more bursty, necessitating a larger buffer size to absorb the burstiness.

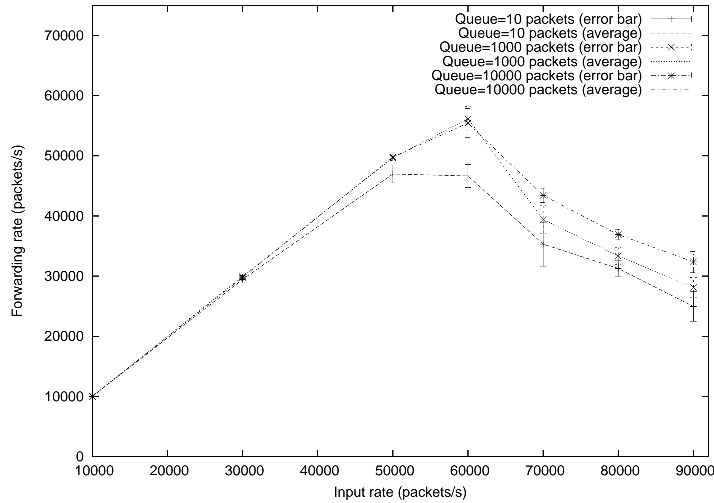

Figure 13: Interrupt mode vanilla IP forwarding rate and percentage with buffer sizes of 10, 1000 and 10000 packets and at different offered 64-byte packet rates.

In an experiment, we measure how the finer preemption granularity proposed in Section 3.1 may impact resource (i.e., buffer) provisioning compared with Click's original mechanism. We configure two flows, A and B. A has only one simple processing element that does little more than queuing each received packet for the output interface. B has the same simple element as A, but in addition n delay elements – each artificially consuming about 1 μs of CPU time – configured into a processing pipeline with no intervening Queue elements. In the original mechanism, the pipeline of $n+1$ elements is not preemptible, but it is preemptible at element boundaries with the proposed changes. We generate 64-byte packet arrivals for the two flows at a rate of about 5200 packets/s. We vary n from 0 to 12 in a set of runs, and report the minimum buffer sizes needed by A to achieve lossless fowarding in the original and new mechanisms, respectively. Fig. 14 shows the results. Notice that for the original mechanism, the required buffer size for A increases roughly linearly as n increases. With fine-grained preemption, however, the required buffer size increases from 1 to 2 as n increases from 0 to 1, but stays at the value 2 as n further in-

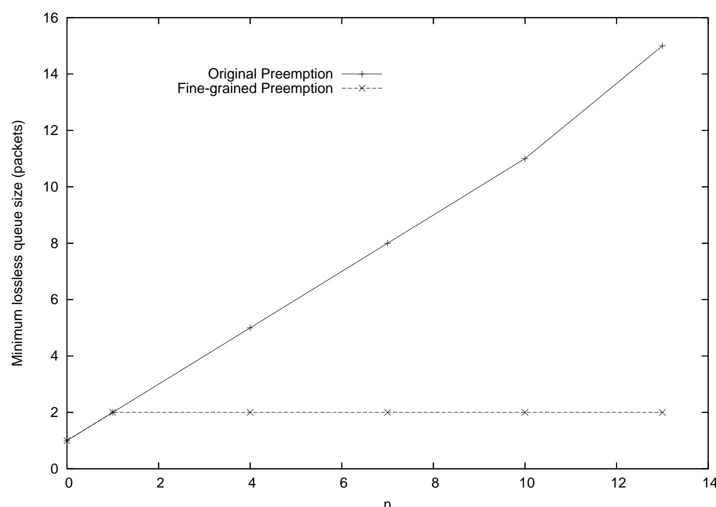

Figure 14: Minimum buffer size for lossless forwarding by flow A, as a function n, the number of delay elements used in competing flow B's pipeline. Original versus fine-grained preemption mechanisms.

creases. Hence, although both mechanisms can assure a long-term forwarding rate for A independent of B's processing pipeline, fine-grained preemption has the added advantage of keeping A's buffer requirement largely unchanged in the different runs.

6.6 Video scaling

Video scaling is designed to respond to network congestion, and is most useful for connections without access to guaranteed link bandwidth. Hence, we do not perform real-time link scheduling in our experiments. Instead, default FIFO packet scheduling is used for each network output port.

The experimental network setup for video scaling is shown in Fig. **??**. In the figure, a wavelet video stream consisting of 300 frames and with a peak bandwidth requirement of 2.6 Mb/s is being sent at 25 frames/s from bolling to madrigal, through the CROSS/Linux router cadiz. The video stream, encoded to have one base layer and 127 enhancement

layers, is displayed at madrigal when received. At cadiz, it competes for resources with a cross traffic stream of UDP packets, sent at different bit rates and requesting different per-flow processing, from sevilla to madrigal. The direct links shown between machines are 10 Mb/s point-to-point ethernet connections. Interrupt I/O is being used.

In the presence of network congestion, CPU allocations have a significant impact on the quality of the video received. In a set of experiments, we run the video flow with a competing UDP flow generated at a rate of 12,499 packets/s (packet size of 64 bytes). Each UDP packet receives CPU-intensive per-flow processing to create CPU congestion. (The actual CPU utilization is 100% throughout each experiment.) When the video flow is routed through the scaling service, we vary the CPU allocation of the flow to be 0.003%, 0.067% and 0.122%, respectively. The remaining CPU capacity, less 20% given to the global router functions, is entirely allocated to the competing UDP flow. Fig. 15 profiles the PSNR of the received video. The average PSNR's for 0.003%, 0.067% and 0.122% of video CPU allocation are 20.56, 21.67 and 22.61 dB, respectively. All 300 frames are displayed for each experiment using video scaling. For comparison, we also show the received video quality with drop-tail and 0.183% CPU allocation to the video flow. In spite of the relatively high CPU allocation, the video quality is very low – only 7 frames are successfully displayed, with an average PSNR of 23.12 dB. We conclude that video scaling, when given a sufficient CPU share to run, can significantly improve the video application's ability to gracefully respond to network congestion.

7 Related Work

Component-based synthesis of network protocols has been advanced in x-kernel [3], and adopted in recent extensible software-based routers [1, 10, 12]. A notable example is

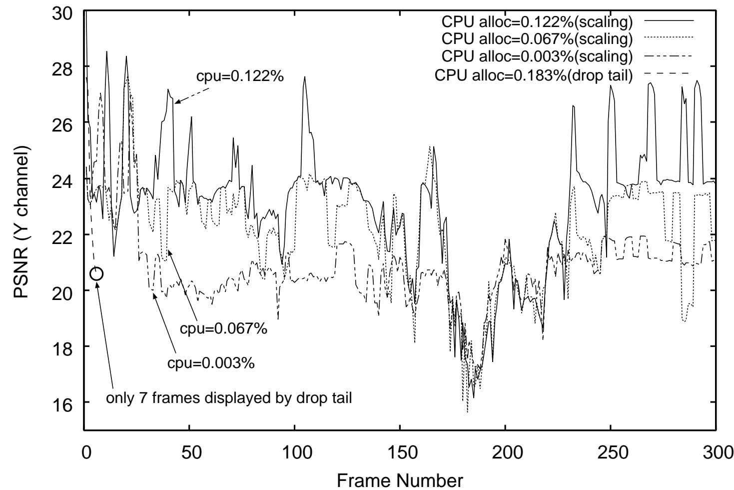

Figure 15: Received video quality with the video scaling service running at different CPU rates, under CPU and network congestion.

router plugins [1] – however, plugin gates are fixed in the IP forwarding path and cannot be dynamically extended. Moreover, the previous work [1, 3, 10, 12] focuses neither on scheduling issues for the software elements themselves nor issues in the context of a complementary service control plane. Our forwarding plane implementation leverages Click [5, 6]. We support the use of Click elements with push/pull data movement as router service components, and exploit Click's configuration language and system support in constructing flow service pipelines. However, Click does not provide the control plane discussed in this paper. Moreover, we have greatly extended Click in many aspects of flow and control plane scheduling.

There has been recent work on resource management in software routers. Qie *et al.* [8] present very interesting experimental results pertaining to balancing between input, output, and per-flow processing in their software router. We have investigated similar issues of CPU balance in our system. However, our focus is on a system that supports configurable routing elements, whereas their system does not provide such support. To

reduce context switching, they use the technique of batching packets. Our system takes a more fine-grained preemption approach that allows a flow's packet to be preempted at element boundaries. Moreover, important features of flow signaling and service extension, and their interactions with the forwarding plane, are not discussed in [8]. CROSS [13] advances a multiresource scheduling architecture based on *resource allocations*. We use resource allocations in system-level scheduling between the forwarding and control planes. However, CROSS is not element-based and, therefore, does not address a lot of the scheduling issues presented in this paper.

Recently, the use of network processors in a software router, chiefly for data plane services, is reported in [10]. By using different processors (general purpose versus specialized) for various data and control plane services, new scheduling problems arise, which is an interesting area for future research.

Lastly, our work complements existing work in resource scheduling algorithms [2, 11]. We have studied more generic QoS issues than the scheduler itself, including per-flow resource acounting, flow preemption granularity, and CPU conservation for idle elements. While we have used SFQ in our experimental evaluation, our work is relevant in the context of diverse schedulers and can be used together with these schedulers.

8 Conclusions

We have presented the CROSS/Linux software router. The router allows more complex router services to be constructed from simpler and well understood building blocks. Moreover, it is truly dynamically extensible through the flow signaling and on the fly service configuration mechanisms. We have examined in detail various issues of QoS provisioning. For the forwarding plane, we discuss flow-based resource scheduling, and

exploit the lightweight nature of elements to support fine-grained preemption of flow packets. We have also studied how buffers should be provisioned to achieve lossless forwarding of packets under conditions of polling versus interrupt, and various CPU balance between input, output and processing. We have evaluated resource contention issues between the forwarding and control planes. Diverse experimental results show that our router can achieve robust lossless forwarding of packets, and can provide QoS support without excessive performance penalty. Finally, we have prototyped and evaluated a video scaling service to demonstrate benefits for end users.

References

[1] D. Descaper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM*, Vancouver, Canada, Sept 1998.

[2] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. 2nd USENIX OSDI*, 1996.

[3] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):64–76, January 1991.

[4] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, and B. Plattner. An active router architecture for multicast video distribution. In *Proc. IEEE Infocom*, March 2000.

[5] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[6] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 217–231, Kiawah Island, South Carolina, December 1999.

[7] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law Internets. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001.

[8] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[9] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[10] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001. to appear.

[11] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proc. ACM SIGCOMM*, September 1997.

[12] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proc. ACM SOSP*, December 1999.

[13] D. K. Y. Yau and X. Chen. Resource management in software-programmable router operating systems. *IEEE Journal on Selected Areas in Communications*, 19(3), March 2001.

[14] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Trans. Networking*, August 1997.

[15] D. K. Y. Yau, J. C. S. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Networking*, February 2005.