Department of Computer Science Technical Reports

Department of Computer Science

2001

Application Performance on the CROSS/ Linux Software Programmable Router

Prem Gopalan

Seung Chul Han

David K.Y. Yau
Purdue University, yau@cs.purdue.edu

Xuxian Jiang

Puneet Zaroo

Report Number:

01-019

APPLICATION PERFORMANCE ON THE CROSS/LINUX SOFTWERE-PROGRAMMABLE ROUTER

Prem Gopalan
Seung Chul Han
David K.Y. Yau
Xuxian Jiang
Puneet Zaroo
John C.S. Lui

CSD TR #01-019
October 2001

Application Performance on the CROSS/Linux Software-programmable Router

Prem Gopalan, Seung Chul Han, David K. Y. Yau, Xuxian Jiang, Puneet Zaroo, John C. S. Lui

Abstract—

We present CROSS/Linux, a software-programmable router platform that combines the resource management capabilities of CROSS from our earlier work and the modular configurability of Click from MIT. By additionally integrating a remote code downloading mechanism and a multihop signaling protocol, CROSS/Linux is dynamically extensible, configurable, and able to provide predictable processing of network flows that require QoS-aware access to multiple resources. We discuss our integration strategy – in particular, flow signaling and the assimilation of flow element scheduling in Click into our resource management framework. CROSS/Linux can support diverse per-flow processing that gives benefits to end users. We present and evaluate two applications: intelligent video scaling in response to network congestion, and router throttling as a defense mechanism against distributed denial-of-service attacks. While intelligent video scaling has been demonstrated in previous work, our focus is on how guaranteed access to system resources can impact scaling performance.

I. Introduction

We target a software-programmable router that is dynamically extensible, configurable, and able to predictably process network flows that require QoS-aware access to multiple resources. To realize our goal, we integrate the resource management capabilities of CROSS [21] from our previous work and the modular configurability of Click [9] from MIT. We name our system CROSS/Linux to reflect the current implementation efforts using fully open source Linux.

In Click, *elements* are C++ kernel modules each implementing a simple router function (e.g., packet classification, queueing, and packet scheduling), which can be composed and configured into per-flow processing pipelines. To fully realize the resource management ability of CROSS, we address the issue of scheduling Click elements with per-flow QoS allocations. A three-level scheduling policy is defined. At the first level, a real-time CPU scheduler [21] allocates CPU resources between a modified Click element scheduler, a *control thread* for on-the-fly flow management (described in Section IV), and all eligible (in the CPU scheduling sense) system and user processes in Linux. Second level scheduling takes place when the Click element scheduler allocates its CPU share among global router functions of network input, network output, and vanilla IP forwarding, and all the backlogged flows implementing per-flow processing. At the third level, a scheduled flow selects one of its eligible elements for execution.

A network input element is responsible for moving arriving packets from a network interface into per-flow input queues, after packet classification. A network output element is responsible for moving packets from their output queues to an outgoing network interface; it may perform link level scheduling to provide per-flow bandwidth guarantees or differentiation. For a device driver operating in

Contact author: D. Yau (yau@cs.purdue.edu); P. Gopalan is now with Mazu Networks, Cambridge, MA (work done while he was at Purdue); S. Han, D. Yau, X. Jiang and P. Zaroo are with the Department of Computer Sciences, Purdue University, West Lafayette, IN; J. Lui is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, Hong Kong. Research supported in part by the National Science Foundation under grant numbers EIA-9806741 and CCR-9875742 (CAREER), and in part by CERIAS

polling mode (as opposed to being interrupt driven), the input and output elements will additionally handshake with the device for the polling operations. Polling can significantly contribute to system efficiency and stability [12], [15].

Linux processes competing with the Click element scheduler for CPU time may belong to the control plane of the router, e.g., routing and signaling daemons. They may also correspond to per-flow processing modules started in user space [21]. As discussed in [21], user-level modules provide benefits of fault containment for untrusted code. These processes can run with specified allocations of CPU time, network bandwidth, disk bandwidth, and physical memory.

Beyond resource management at a single router, we support resource allocation at designated hops on an end-to-end network path. The inter-machine signaling protocol is sender-driven. A signaling packet is IP encapsulated with source routing through specified intermediate hops. An *activate* signaling packet is interpreted for flow setup with resource allocation at all CROSS/Linux routers, and is passed uninterpreted by a non-cognizant router. Flows establish *soft state*, which can be deallocated either through timeout or an explicit *deactivate* signaling packet.

We prototype applications on CROSS/Linux to gain experience with the cost and benefit of various per-flow processing functions. Our current focus areas are real-time multimedia data streaming and network security. The value of in-network processing for mutlimedia streaming is increasingly well documented (e.g., [3], [7], [8]), although some of these services can also be deployed as *active services* [7] using proxy servers on a network path.

For security, we are motivated by the observation that router software plays a critical role in ensuring the "health" of a public network infrastructure. First, since security problems are highly subtle, parts of the running kernel – in spite of its "maturity" – may have obscure security bugs, making the routers targets for attacks. Patches to fix newly discovered security bugs must be applied in a timely manner. Second, security loopholes have been discovered for main stream routing protocols (e.g., see proposals for securing OSPF and distance vector routing [4], [13]), which should be removed in newer protocol versions. Third, as new forms of network attacks appear, new defense mechanisms [14], [16], [17], [10] may be designed and deployed to improve the security of a network. We believe it is highly useful to have a scalable and automatic deployment mechanism to deliver bug fixes and new security services to a large number of routing points, with minimal human intervention and disruptions on existing service.

A. Our contributions

The importance of flow-based scheduling is widely recognized for providing performance isolation between heterogeneous router services which have QoS constraints or do not necessarily trust each other [5], [21]. Meanwhile, the value of modular configurability of router services is recently shown [9]. We advance a working system that combines these important features. Together with our flow signaling and *on-the-fly* service extension mechanisms, we demonstrate a software-based router that can provide value-added services to users in a QoS-aware manner, and promises to keep up with the evolving Internet with greater facility than existing systems. Our experience building and evaluating two useful value-added services to users (interesting in their own right) gives new insights about these applications, and demonstrates the utility of our system.

B. Paper organization

The balance of the paper is organized as follows. We briefly review the Click modular router architecture in Section II. In Section III, we detail the design of per-flow resource scheduling in CROSS/Linux. The use of a signaling mechanism to create new flows and configure them on-the-fly is discussed in Section IV. In Section V, we overview two router services prototyped on CROSS/Linux: router throttle as a defense against DDoS attacks and intelligent video scaling in response to network congestion. Section VI presents performance results for the applications running on our platform. In particular, we carefully study the effects of CPU allocation and interrupt versus polling I/O on video scaling performance. Section VII discusses related work. Section VIII concludes.

II. Background

For service configuration, we leverage the Click modular router. For completeness, we review the Click router architecture. Further details can be found in [9]. 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 interace, packet classification, queueing, 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. Hence, in general, packets flow along the edges of the flow graph, from input to output. They will receive customized protocol processing according to the actual paths they traverse.

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 from 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*.

Click has to schedule the execution order of elements. 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. When that happens, 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 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.

III. Single node resource scheduling

Resource management in CROSS is based on *resource allocations* [21]. Resource allocation objects allow router services to have QoS-aware access to various system resources, including CPU time, network bandwidth, disk bandwidth and memory share. This translates into predictable performance for user services, on a per-flow basis.

Click processes elements in the context of a designated control thread. Each element can be given a number of stride scheduling tickets [19], which determines the element's share of the CPU. Elements can be put on a *task queue* as a policy decision [9]. Among all the elements appearing in the task queue, the scheduler selects the next one for processing according to their stride scheduling priorities. This allows to balance various *push/pull* processing between different input/ouput interfaces. However, since the same element can be used by multiple flows, element-based scheduling is not compatible with the per-flow resource scheduling paradigm in CROSS.

We modify the Click element scheduler to support *per-flow* scheduling. Each flow is defined by a *flow specification* (e.g., 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) installed with the packet classifier. It prescribes a processing pipeline of elements, which is assumed fixed for the flow's life time. Unlike Click, we allow individual flows to be given their own resource allocations. In addition, since multiple elements for the same flow can be eligibile for running – due to the presence of *Queue* elements in Click [9] – we allow a flow to in turn apportion its resource allocation among the constituent elements. (Hence, a flow element may have scheduling state in the context of its flow.) With the modifications, the Click's task queue contains a set of all eligible *flows* in CROSS/Linux – as opposed to elements in the original design. Each eligible flow is represented by an fRouter object in the task queue.

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 ports, and an element for vanilla IP forwarding, are important examples. We treat these global elements as belonging to certain *global "flows"* (each represented in the task queue by an ioRouter object). For the purpose of scheduling, global flows are just like normal flows. They can be endowed with specified resource allocations, thus allowing their elements to compete for system resources with other per-flow elements. The assignment of gloabl 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.

To support accurate per-flow scheduling, early demultiplexing of packets into their flows is necessary. This is done by placing a Click Queue element immediately after the network input element associated with an input port (see Section II and [9]). This allows the input element to return immediately after moving a packet from the input port and classifying the packet into the appropriate flow, instead of performing any per-flow processing in the input element's global context. (In essence, this solves the problem of *hidden scheuling* [22].)

The per-flow fRouter object contains a set of eligible elements and their respective current packets for the flow. One of these elements, called flowStart, is responsible for initiating per-flow processing of packets arriving for the flow. Once scheduled, fRouter selects the next element to run based on the (flow-private) scheduling state of these elements. Fig. 1 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.

IV. Flow Signaling and Service Configuration

Section III describes flow-based 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 processing elements. Such flow management is controlled 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 CROSS/Linux packet classifier reading from an input port identifies these control packets and delivers them to a control queue. The control queue is processed in FIFO order by a CROSS/Linux *control thread.* The control thread receives its CPU share from the *first-level* CPU scheduler and, as such, competes directly for system resources with the flow scheduler described in Section III. The control thread performs flow management by running a new Click object called FlowManager (the *flow manager*). FlowManager is a derived class of the original IPFilter element (corresponding to a packet classifier) in Click. Compared with IPFilter, it has the added flexibility of being extensible with new ports and filter rules, key to on-the-fly flow setup.

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 newly created Queue element, called flowQueue, corresponding to the new flow. Finally, the flowQueue element is used to initialize an fRouter for the new flow with the flowStart element described in Section

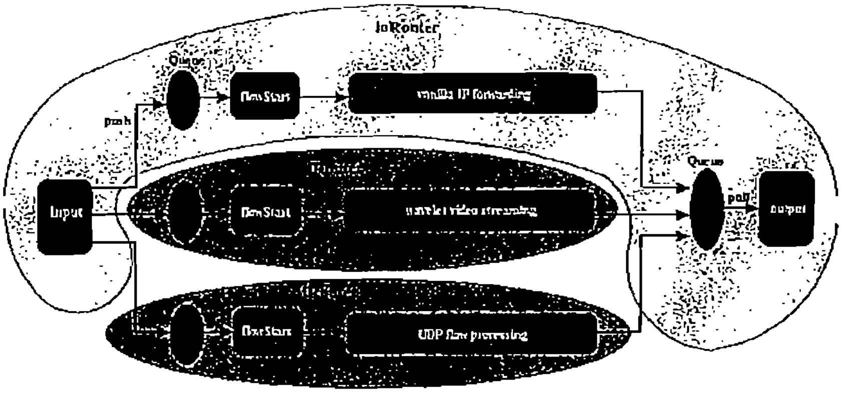

Fig. 1. A CROSS/Linux router configuration with one global flow and two user flows.

III. The fRouter is allocated system resources according to parameters in the IC_SETUP packets. Later packets that match the filter rule for the new flow are then delivered to the corresponding flowQueue. These packets will be picked up by flowStart when the corresponding fRouter is selected to run.

Flow configuration. An IC_CONFIG control packet is used to add/delete an element from the processing pipeline 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 a user-level active network daemon anetd [] to download the named service from a remote node.[1] The anetd implementation 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. Finally, 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.

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.

V. Applications

We are building user applications on CROSS/Linux. We describe two services: router throttling as a defense mechanism against distributed denial-of-service (DDoS) attacks, and wavelet video scaling for application-aware network congestion control. The services are implemented in C++ as Click elements. They are compiled as Linux loadable kernel modules for deployment.

A. *Router throttle*

Router throttling [10] is a resource management based defense mechanism against DDoS attacks (e.g., [1], [2]). Its goal is to protect a server system from having to deal with excessive service request arrivals (from a cohort of attacking machines) over a global network. To do so, a *proactive*

[1] We originally implemented a CORBA based naming and code download service, with the download process supervised by the control thread. We decided in favor of anetd because of simplified synchronization and its wider acceptance in the active network community.

approach is used: Before aggressive packets can converge to overwhelm a server, we ask routers along forwarding paths to regulate the contributing packet rates to more moderate levels, thus forestalling an impending attack. The basic mechanism is for a server under stress (e.g., being flooded with attack traffic), say S, to install a *router throttle* at an upstream router several hops away. An installed throttle limits the rate at which packets destined for S will be forwarded by the router. To accomodate bursty traffic, a throttle is implemented as a leaky bucket with the desired rate limit and some bucket size s (in bits) to absorb the burstiness. Traffic that exceeds the rate limit can either be dropped or rerouted to an alternate server.

In a related technical report [10], we study the problem of determining appropriate throttle rates at distributed routing points, such that, globally, S exports its full service capacity to the network, but no more. The "appropriate" throttles are adaptive to the current demand distributions, and are negotiated dynamically between server and network. Via simulations, we show that router throttling can offer significant relief to a server that is being flooded with malicious attacker traffic. First, for aggressive attackers that send at significantly higher rates than legitimate users, the throttle mechanism can preferentially drop attacker traffic over good user traffic. This allows a much larger fraction of good user traffic to make it to the server as compared with no network protection. Second, for both aggressive and "meek" attackers (i.e., attackers that send comparable amounts of traffic as legitimate users), throttling can regulate the experienced server load to within its design load limits, so that the server can remain operational during a DDoS attack. The ability to increase the availability of a Web server during attack episodes is also demonstrated through simulations.

In this paper, we prototype router throttling on CROSS/Linux. The implementation complements our simulation results, and allows us to measure the deployment costs of the mechanism at a network node. We are, for example, interested in the memory and processing requirements of throttling as a function of the number of throttles installed. More generally, the implementation exercise demonstrates the ability of CROSS/Linux to dynamically extend the security features at a router. Other security mechanisms (e.g., [14], [16], [17]) useful in diverse scenarios can similarly be introduced in a seamless manner using CROSS/Linux.

In the implementation, a server, say S, requests throttling at a CROSS/Linux router by sending it a control packet. The control packet specifies the IP address of S, and the throttle leaky bucket size and token rate. On receiving such a packet, CROSS/Linux checks if the throttle service is already available at the local node. If not, it uses the service downloading mechanism in Section IV to fetch the throttle code from a designated code server, and links the code dynamically into the running kernel. When the throttle service has been linked to the kernel, it is configured into the processing pipeline of packets destined for S. A configured throttle limits the long-term forwarding rate of packets for S to the token rate, and the maximum burst size to the leaky bucket size. Any excess packets are *dropped* in our implementation.

B. Video scaling

A media scaling service is reported in [8] for router plugins [5]. 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. Like router throttle, it can be fetched and loaded on demand, in response to user requests. While the same service has been demonstrated in [8], our goal in this paper is to understand how resource allocation in CROSS/Linux can impact video quality perceived by end users. In particular, video scaling requires sufficient *processor 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.

VI. Experimental Results

We present experimental results to illustrate application performance on CROSS/Linux. The routing platform used is a Pentium III/864 MHz PC fitted with four PCI 3Com 3c59x (vortex) 10/100 Mb/s ethernet interfaces. We made our own changes to the vortex device driver to support polling I/O.

For the global router functions, we schedule them in the context of a *single* global flow. I.e., one ioRouter object ran on an experimental router. We used Click's default algorithm to adaptively allocate the global flow's CPU allocation to the individual flow elements. For example, when Click sees a burst of arriving packets, it will automatically increase the CPU share given to the relevant network input element. Because of the design in CROSS/Linux to isolate the resource allocations between flows, any such rate increase occurs only within the context of the global flow in our system.

A. Service extension

We measure the overhead of configuring and integrating new router services in CROSS/Linux, as described in Section IV. In the experiments, the machine cadiz shown in Fig. 2 is the CROSS/Linux router on which the new services are to be installed. It runs in our research lab in the Purdue CS department. The implemented code is not initially available locally at cadiz, and has to be fetched from ponce (see Fig. 2), a web server owned by the campus computation center, and connected to cadiz via the public campus Internet. Therefore, the experiments give an idea of the kind of performance when code may have to be fetched from remote servers accessed through a typical shared network infrastructure.

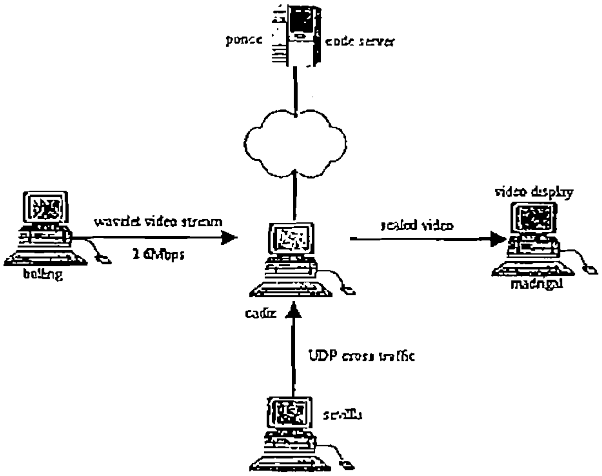

Fig. 2. Experimental network setup for video scaling, with a remote code server accessed through the Internet.

Fig. 3 shows the HTTP transfer times for anetd to obtain the code from ponce's web server, as a function of the code size. The figure plots the average transfer time over 50 measurements for each service, and the standard deviation is shown as an error bar. Notice that the average transfer time is largely directly proportional to the code size. The video scaling service implemented as wavescale.o has size about 9.8 kbytes, and requires a transfer time of about 23.19 ms. Router throttling implemented as throttle.o has size about 10.5 kbytes, and requires a transfer time of about 23.83 ms. Fig. 4 reports the time taken to dynamically link wavescale.o and throttle.o, respectively, into the running Linux kernel. The time to link our modified Click module (click.o) is also shown for comparison. From the figure, wavescale.o and throttle.o each takes about 10 ms of linking time, whereas click.o, being larger and more complex, takes about 80 ms. Lastly, the time taken to configure the video scaling and router throttle services into their corresponding processing pipelines is measured to be 11.31 and 11.06 ms, respectively.

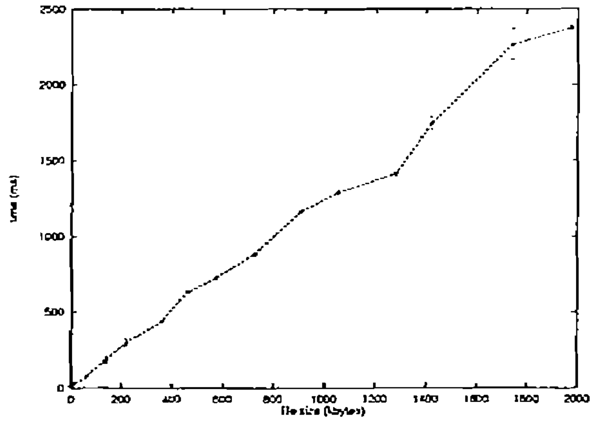

Fig. 3. A plot of service module transfer time using HTTP, as a function of the code size.

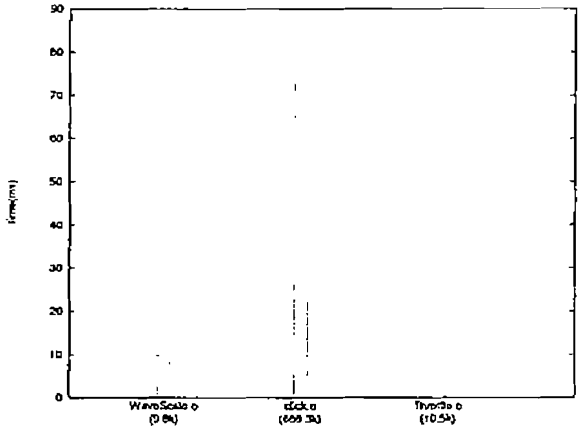

Fig. 4. Dynamic linking time for the video scaling service, the modified Click module, and the router throttle service.

B. Resource implications for video scaling

Video scaling is designed to respond to network congestions, 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. 2. 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.

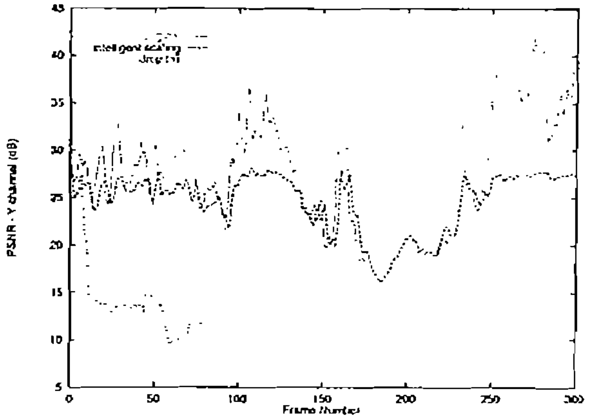

Fig. 5. Received video quality for video scaling versus drop-tail, under network congestion.

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. Unless otherwise stated, the direct links shown between machines are 10 Mb/s point-to-point ethernet connections.

B.1 Network congestion

In a set of experiments, we verify the value of intelligent video scaling during network congestion. We constrain the outgoing link bandwidth from cadiz to madrigal to be 8 Mb/s. We run the video flow in competition with a UDP flow. The UDP flow is being generated at a rate of 10,000 packets/s, with packet size of 64 bytes. Interrupt I/O is being used. Fig. 5 profiles the PSNR of the video displayed at the receiver machine, with and without video scaling at the router. With video scaling, all 300 frames are displayed at the receiver, with an average per-frame PSNR of 24.6 dB. With drop-tail, the indiscriminate drops cause loss of playback synchronization at the receiver, and only 79 frames are successfully displayed. The average PSNR is 14.36 dB.

B.2 CPU congestion

Next, we examine the effects of CPU allocation on video quality at the receiver. In a set of experiments, we vary the CPU rate allocated to the video flow to be 0.006%, 0.061%, 0.091%, and 0.122%, respectively. A CPU allocation of 20% is given to the global router functions of input, output, and

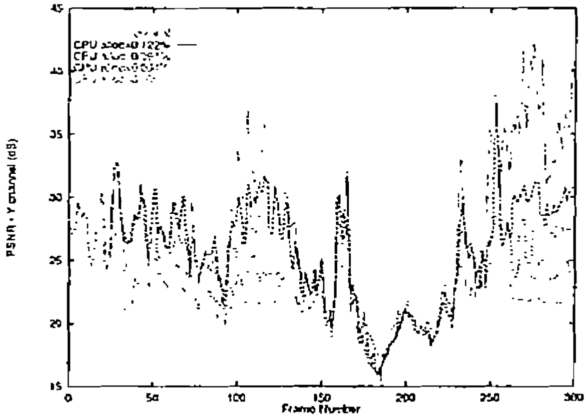

Fig. 6. Received video quality with the video scaling service running at different CPU rates, under CPU congestion.

vanilla IP forwarding. These global functions are not CPU intensive in the experiments, and do not use up their CPU allocations.[2] The remaining CPU capacity is entirely allocated to a competing UDP flow. We run the competing UDP flow at a low bit rate, so that the network is not congested. However, we performed CPU-intensive per-flow processing, artificially created to cause CPU congestion, for each UDP packet. The actual CPU utilization is 100% throughout each experiment. Figure 6 profiles the video PSNR at the receiver. Notice that in the face of competition from the UDP flow, the amount of CPU time guaranteed to the video flow has a significant impact on the receiver video quality. The average PSNR's for 0.006%, 0.061%, 0.091%, and 0.122% of the allocated video CPU rate are 21.70, 23.06, 24.94, and 25.71 dB, respectively.

The loss in video quality is due to packet loss. We measure the number of packets dropped at the queues linked to the flowStart element of video scaling and the queue linked to the network output element, respectively. Since there is no network congestion, we observe negligible packet loss at the network output queue. For loss at the video queue, Fig. 7 shows the total number of packets dropped for the entire video as a function of the allocated CPU rate to the video flow. The results confirm that a sufficient CPU rate is needed to allow the video flow to process its packets fast enough, in order to avoid buffer overflow at its input queue.

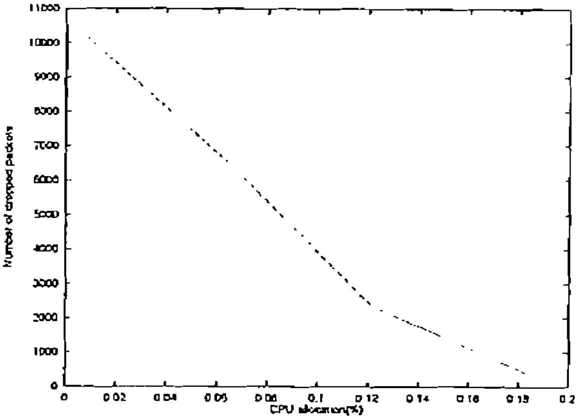

Fig. 7. Total number of video packets dropped at the video flow queue as a function of the allocated CPU rate to the video flow.

B.3 CPU and network congestion

In the presence of network congestion, CPU allocations similarly have a significant impact on the quality of the video received. In this 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 the 20% given to the global router functions, is entirely allocated to the competing UDP flow. Fig. 8 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.

[2] In our experiments, we route small packets at a rate of about 10,000 packets/s. Even including interrupt overhead, the maximum forwarding rate and loss-free forwarding rate of 64-byte packets on our platform is about 65,000 and 50,000 packets/s, respectively (using interrupt I/O).

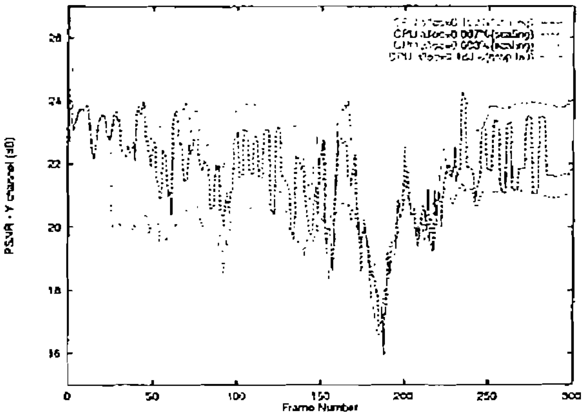

Fig. 8. Received video quality with the video scaling service running at different CPU rates, under CPU and network congestion.

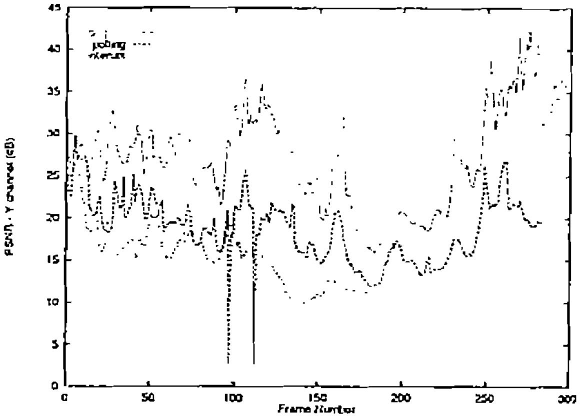

Fig. 9. Received video quality for router employing polling I/O versus interrupt I/O, with UDP cross traffic generated at a rate of 9,500 packets/s (packet size 64 bytes).

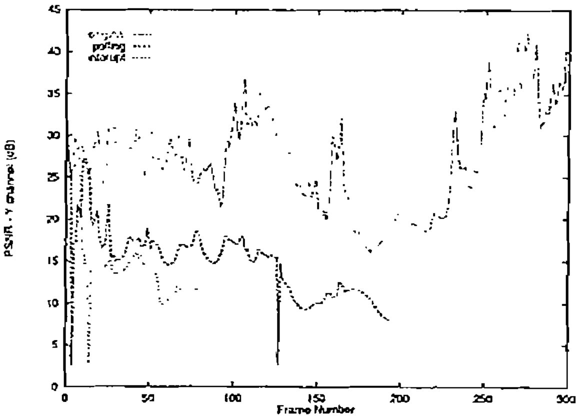

Fig. 10. Received video quality for router employing polling I/O versus interrupt I/O, with UDP cross traffic generated at a rate of 10,000 packets/s (packet size 64 bytes).

B.4 Polling I/O

While the previous experiments used interrupt I/O, it has been shown that polling I/O can give significantly improved system performance when forwarding high-rate traffic. This is because polling does not incur expensive per-packet overhead of interrupt handling. To demonstrate the effect of polling versus interrupt I/O *on our streaming video application*, we route the video flow and a competing UDP flow under either configuration. No scaling service is employed for the video flow. In a first experiment, the UDP flow is generated at a rate of 9,500 packets/s, with packet size of 64 bytes. Each UDP packet receives normal IP forwarding. Fig. 9 compares received video qualities for polling versus interrupt I/O. With polling, 282 frames are successfully displayed at the receiver, with an average PSNR or 18.68 dB. With interrupt, 181 frames are successfully displayed, with an average PSNR of 15.63 dB. Notice that the PSNR profile of polling is consistently better than that of interrupt. The original PSNR profile, with an average of 27.2 dB, is also shown for comparison.

In another experiment, we increase the competing UDP flow rate to 10,000 packets/s, while keeping the packet size at 64 bytes. Figure 10 profiles the received video qualities for polling and interrupt. With polling, 192 frames are successfully displayed at the receiver, with an average PSNR of 14.96 dB. With interrupt, 77 frames are displayed, with an average PSNR of 14.69 dB. We conclude that the increased efficiency of routing packets by polling I/O translates into significant gains in video quality at the receiver.

C. Router Throttle

To measure the memory overhead of router throttle, we first load the CROSS/Linux router and the throttle modules into the kernel. Then, using the /proc file system, we note the amount of memory allocated as 540 kbytes. We then install up to 1000 throttles one by one, observing the increase in memory allocated after each throttle installed. Figure 11 plots the average memory allocated, as a function of

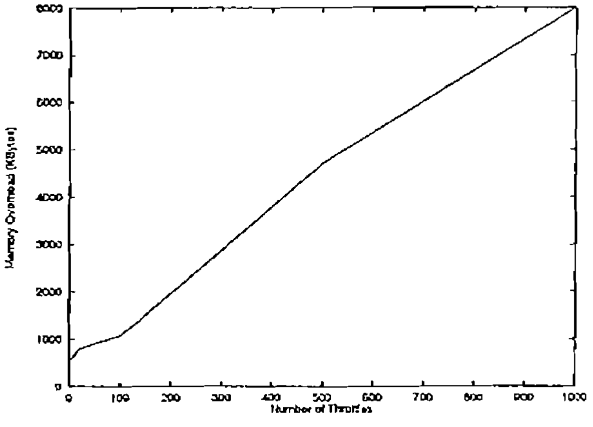

Fig. 11. Router throttle memory overhead, as a function of the number of throttles installed.

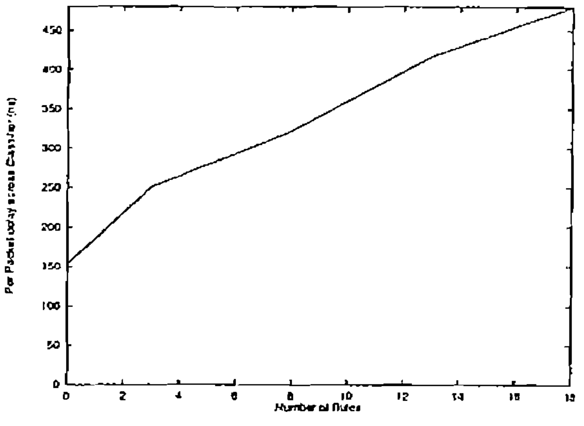

Fig. 12. Delay performance of router throttling, as a function of the number of throttles installed

the number of throttles installed, over several experiments. The results show that the memory allocated increases largely linearly with the number of throttles, with an average per-throttle memory of about 7.5 bytes.

We breakdown the delay of throttling into two components: throttle lookup in the packet classifier, and the delay due to the throttle element itself. We found that the delay through the throttle element is about 200 ns, independent of the number of throttles installed. This small and relatively constant delay is very encouraging, showing that throttling is not inherently expensive. Throttle lookup depends heavily on the performance of the packet classifier. We use the default classifier in Click. From Fig. 12, notice that the "base" classifier delay (i.e., without any created flows) is about 150 ns. Following that, the delay increases about linearly with the number of throttles installed, reaching about 475 ns for 18 throttles. We expect that by porting our previous classifier in [21] – shown to have highly scalable lookup performance – to CROSS/Linux, we can much improve upon the linear increase in delay.

To ascertain how the throttle overhead affects throughput, we measure the maximum achievable forwarding rates of packets through CROSS/Linux, with no throttled flow, to up to 18 flows created for throttling. Fig. 13 shows the average number of 64-byte packets we can forward per second, as a function of the number of throttled flows.

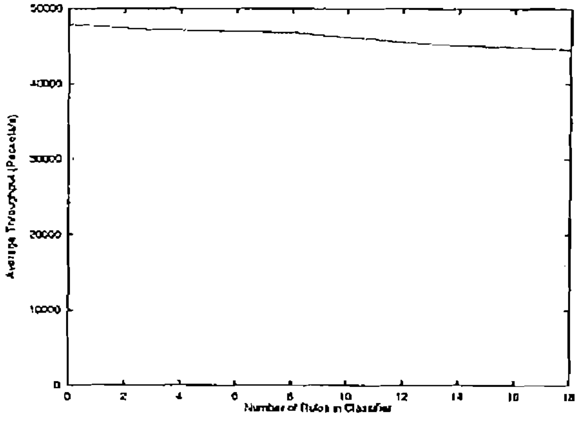

Fig. 13. Throughput performance of router throttling, as a function of the number of throttles installed

VII. Related work

Component-based synthesis of network protocols has been advanced in x-kernel [6], and adopted in recent extensible software-based routers [18], [5], [20]. A notable example is router plugins [5] – however, plugin gates are fixed in the IP forwarding path and cannot be dynamically extended. Our work heavily leverages the Click router [12], [9]. 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, we have extended Click in several important directions. First, we run Click in the context of the CROSS resource management framework, which allows multidimension QoS-aware resource allocation at the level of processes and threads. Hence, the scheduling of flow service elements can be controlled in relation to other system activities, such as routing in the control plane. Second, we have adapted element scheduling in Click to a *per-flow* paradigm, key to providing performance isolation between users and applications. Third, we provide a signaling mechanism to create flows with given resource specifications on-the-fly, and to incrementally extend or modify a flow processing pipeline.

Resource management in software-programmable routers, especially for both computation and forwarding resources, is an important problem. However, relatively little work has been done in the area. Qie et al [15] present very interesting experimental results pertaining to balancing between input, output, and flow processing in a software router. Our experiments in this paper have stressed resource contention during flow processing. In our system, scheduling control between input, output and flow processing can be specified in various ways. For example, one can define a global flow each for network input and output, and give these flows certain resource shares relative to other flows in the system. Alternatively, it is possible to use one global flow and assign different resource shares to the input, output and vanilla IP forwarding elements within the flow. Currently, we use the single global flow approach, with Click's default adaptive resource allocation policy between the flow's elements. CROSS [21] advances per-flow multiresource allocation and scheduling for router services. We extend CROSS to include service extensibility and configurability inside the kernel. Our investigation on polling I/O follows earlier work to eliminate received livelocks in an OS [11]. The polling I/O approach is also adopted in [9], [15].

Recently, the use of network processors in a software router, chiefly for data plane services, is reported in [18]. 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.

The video scaling service we use has been reported in [8], but without reference to the effects of resource scheduling on application performance. We demonstrate our system's ability to support video scaling on-the-fly, and carefully study the relevance of resource management in CROSS/Linux to the effectiveness of video scaling. Router throttling is described in a related technical report [10]. We complement the simulation results in [10] by measuring the deployment costs of router throttling on a software-based router.

VIII. Conclusions

We presented the CROSS/Linux software-programmable router. Our router integrates the resource management capabilities of CROSS [21] and the service configurability of Click [9]. We described our flow signaling mechanism that allows new flows to be dynamically created with on-the-fly service instantiation and configuration. For resource management, we employ a design that allows resources to be scheduled among (i) global router functions of input, output, and vanilla IP forwarding, (ii) per-flow user processing, and (iii) other Linux processes and threads (e.g., routing and signaling daemons, and the control thread for flow signaling). We detailed our design to provide *per-flow* resource allocation in the context of Click elements. We also presented flow signaling to dynamically create flows with given resource specifications, and to configure the flows on-the-fly with new services, possibly fetched from a remote server.

We presented two router services that have been prototyped on CROSS/Linux. For router throttling, we measured its deployment costs on our router platform. This complements previous simulation results that assess the effectiveness of router throttling in countering DDoS attacks in a global network. For wavelet video scaling, we demonstrated how resource scheduling can significantly impact the quality of received video in the face of CPU congestion, network congestion, or both. We believe that CROSS/Linux is an effective platform for providing flexible value-added services sented flow signaling to dynamically create flows with given resource specifications, and to configure the flows on-the-fly with new services, possibly fetched from a remote server.

We presented two router services that have been prototyped on CROSS/Linux. For router throttling, we measured its deployment costs on our router platform. This complements previous simulation results that assess the effectiveness of router throttling in countering DDoS attacks in a global network. For wavelet video scaling, we demonstrated how resource scheduling can significantly impact the quality of received video in the face of CPU congestion, network congestion, or both. We believe that CROSS/Linux is an effective platform for providing flexible value-added services to network users, with useful QoS-aware resource sharing.

ACKNOWLEDGEMENT

S. Jeyaraman implemented polling I/O for the vortex fast ethernet device driver used in our experiments.

REFERENCES

[1] TCP SYN flooding and IP spoofing attacks. CERT Advisory CA-96.21. available at http://www.cert.org/.

[2] Smurf IP denial-of-service attacks. CERT Advisory CA-1998-01, January 1998. avaliable at www.cert.org/advisories/CA-98.01.html.

[3] K. L. Calvert, J. Griffioen, A. Sehgal, and S. Wen. Building a programmable multiplexing service on concast. In *Proc. IEEE ICNP*, Osaka, Japan, November 2000.

[4] S. Cheung. An efficient message authentication scheme for link state routing. In *Proc. 13th Annual Computer Security Applications Conference*, San Diego, CA, December 1997.

[5] 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.

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

[7] S. K. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. *IEEE Network*, February 2000.

[8] 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.

[9] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[10] F. Liang, D. K. Y. Yau, and J. C. S. Lui. On defending against distributed denial-of-service attacks with server-centric router throttles. Technical Report TR-01-008, Dept of Computer Sciences, Purdue University, West Lafayette, IN, May 2001.

[11] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. 1996 USENIX Technical Conference*, 1996.

[12] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans 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.

[13] S. Murphy and M. Badger. Digital signature protection of the OSPF routing protocol. In *Proc. Internet Society Symposium on Network and Distributed Systems Security*, San Diego, CA, February 1996.

[14] 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.

[15] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

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

[17] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. Timothy. Hash-based IP traceback. In *Proc. ACM SIGCOMM*, San Diego, CA, August, 2001.

[18] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Sympo-*

tees. *IEEE/ACM Transactions on Networking*, 6(6), December 1998.