

Processor Scheduler for Multi-service Routers

Ravi Kokku[‡] Upendra Shevade[†] Nishit Shah[†] Ajay Mahimkar[†] Taewon Cho[†] Harrick Vin[†]
NEC Laboratories America, Inc.[‡] The University of Texas at Austin[†]
ravik@nec-labs.com[‡], {upendra, nishit, mahimkar, khatz, vin}@cs.utexas.edu[†]

Abstract

In this paper, we describe the design and evaluation of a scheduler (referred to as Everest) for allocating processors to services in high performance, multi-service routers. A scheduler for such routers is required to maximize the number of packets processed within a given delay tolerance, while isolating the performance of services from each other. The design of such a scheduler is novel and challenging because of three domain-specific characteristics: (1) difficult-to-predict and high packet arrival rates, (2) small delay tolerances of packets, and (3) significant overheads for switching allocation of processors from one service to another. These characteristics require that the scheduler be agile and wary simultaneously. Whereas agility enables the scheduler to react quickly to fluctuations in packet arrival rates, wariness prevents the scheduler from wasting computational resources in unnecessary context switches. We demonstrate that by balancing agility and wariness, Everest, as compared to conventional schedulers, reduces by more than an order of magnitude the average delay and the percentage of packets that experience delays greater than their tolerance. We describe a prototype implementation of Everest on Intel's IXP2400 network processor.

1. Introduction

In this paper, we describe the design and evaluation of a novel scheduler for *high-performance, multi-service routers*. Over the past decade, new uses and commercialization have pushed the Internet well beyond the expectations of its designers. Yet, the network architecture and the services provided by the Internet have changed little. The size and the multi-provider nature of the Internet have led to its ossification [4, 25].

It has been argued [4, 25] that two recent trends—overlay networks and high-performance programmable router hardware—can be combined to address Internet ossification. Overlay networks enable the deployment of several rich network services (such as content distribution [1], virus and worm detection [41], DDoS detection and prevention [38], IPv4-IPv6 translation [36], QoS [35], etc) on the existing Internet. Whereas, high-performance pro-

grammable routers can support multiple such services at multi-Gb/s speeds [17]; these routers utilize multiple parallel processors to achieve high throughput. The two trends together promise the evolution of a new generation of *multi-service routers* [4, 5, 25].

This paper focuses on one key problem in such multi-service routers—*how should processors be multiplexed among competing services such that (1) the number of packets served within their delay tolerance is maximized, and (2) the throughput guaranteed to a service is not affected by traffic fluctuations for other services*. This problem is similar to scheduling in various domains (e.g., real-time systems [8, 11, 23, 31, 34], conventional operating systems [37], web servers [6, 12, 39], software-based routers [27, 32], storage systems [18, 24], and multi-media servers [16, 40]). However, the following unique characteristics of the domain of network applications render the above solutions unsuitable for multi-service routers.

- The arrival rate of packets at multi-service routers can be at least two to three orders of magnitude greater than the arrival rate of requests in other hosting environments (e.g., millions of packets per second vs. thousands of requests per second in a web hosting environment). Further, the minimum inter-arrival time of packets is much smaller than the packet processing time ($t_{iat} \ll t_{pkt}$), and network traffic is bursty and difficult to predict, especially at fine timescales (10-100s of milliseconds) [26, 42].
- Network services are required to process packets within a small delay from their arrivals. In contrast to conventional OSEs and hosting environments where a relatively large delay (e.g., 10ms) is tolerable even for interactive applications, in this environment adding even a 1ms delay per hop (router) for each packet can be prohibitive.
- The context-switch overhead t_{sw} of reallocating processors from one service to another can be large and often one to three orders of magnitude greater than the time required to process a packet. For instance, researchers have observed that simply loading the instruction store of a processing core with the code for a new service on Intel[®]'s network processors takes of the order of a millisecond [21, 9], while a packet can often be pro-

cessed within a few microseconds ($t_{sw} \gg t_{pkt}$). Similarly, Qie et al. [27] observe t_{sw} to be twice t_{pkt} in PC-based routers.

In this paper, we describe the design and evaluation of Everest, a novel scheduler for multi-service routers that meets these challenges by balancing two key properties: agility and wariness. Whereas agility enables Everest to react quickly to fluctuations in packet arrival rate, wariness prevents Everest from wasting computational resources in unnecessary context switches. Everest balances agility and wariness by modeling the delay tolerance of packets and the context-switch overhead explicitly in making processor scheduling decisions. Through a detailed simulation study, we demonstrate that Everest, as compared to conventional schedulers, reduces by more than an order of magnitude the average delay and the percentage of packets that experience delays greater than their tolerance. We then describe a prototype of Everest on Intel[®]'s IXP2400 network processor, and demonstrate Everest's effectiveness in a real multi-service router that hosts two services—a port-scan detector and a TCP-SYN flood detector.

The rest of the paper is organized as follows. Section 2 presents the system model and the requirements of a scheduler in a high-performance multi-service router. In Section 3, we develop a taxonomy of multi-processor schedulers that exposes the drawbacks of existing schedulers and guides the design of Everest. In Sections 4 and 5, we design and evaluate Everest. Section 6 describes the efficacy of Everest in a prototype multi-service router built using the Intel[®]'s IXP2400 network processor. Section 7 summarizes our contributions.

2. System Model and Requirements

Consider a multi-service router that multiplexes N_p processors among S_A services (see Figure 1 and Table 1). Each processor has a local instruction cache to load the services; at any instant, a subset of the processors in the system are allocated to each service. For simplicity, we assume that all processors have the same processing capacity, and at most one service can be loaded and run on a processor¹. Switching a processor's context from one service to another incurs an overhead of t_{sw} time units²; and the processor is unavailable to process packets for that duration. This overhead includes the time it takes to checkpoint and migrate the data of

¹Everest can be extended with a few modifications (by appropriately scaling the throughput that each service receives from the processor and applying scheduling techniques such as those explored by Qie et al. [27] for hosting multiple services on a single processor) to accommodate multiple services per processor and processors with different capacities.

²Although we assume t_{sw} to be the same for all services for simplicity, our solution remains the same when t_{sw} is different for each service.

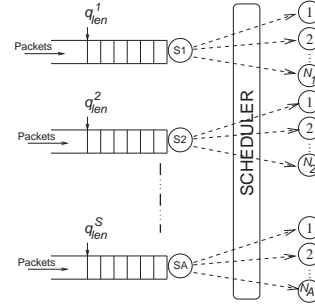


Figure 1. System model.

the service, and to load the code of the service onto a processor's instruction cache. We note here that the instruction caches of processors are often limited in capacity that only a subset of the services can be loaded onto each processor, and hence the above definition of t_{sw} is justified. This hardware model closely approximates Intel[®]'s IXP2400 network processor, a multi-core processor commonly used to design multi-service routers [17].

Each service is associated with a queue; packets are placed in the queue until a processor allocated to the service becomes available to process the packet. Let the time taken to process a packet by service i be t_{pkt}^i units. Packets at each service are processed in the order of their arrival. After being processed by a service, the packet is queued either for processing at another service or for transmission at an outgoing link. We do not deal with the complementary problem of packet scheduling at the outgoing link—several research efforts in the past have dealt with the problem [14, 33].

Let D^i denote the packet delay tolerance for service i , i.e., a packet should be processed by service i within D^i time units of the packet's arrival. Processing packets prior to their delay tolerance at each router ensures that (1) the total delay and jitter perceived by packets that traverse multiple routers between two communicating end-points is within a range acceptable by the end-points, and (2) the queue (buffer) sizes maintained at the routers (services) are small and do not overflow. Also, let $R_{arr}^i(\Delta)$ denote the rate of packet arrivals into the queue for service i , estimated over Δ units of time.

In such an environment, a scheduler that multiplexes processors among the services should meet two requirements: (1) maximize the number of packets processed within the delay tolerance D^i , and (2) isolate the throughput guaranteed to a service from the fluctuating requirements of other services. We define the throughput isolation requirement precisely as follows. Let N_{reqd}^i denote the number of processors that service i needs to process all packets queued at service i at time t within their delay tolerance D^i . Then, at time t , the minimum number of processors that are guaranteed to be allocated to service i (regardless of the requirements of other services) is given by:

$$N_{min}^i = \min(N_{reqd}^i, G^i \times N_p) \quad (1)$$

Table 1. System model parameters for Everest design.

Parameter	Description	Parameter	Description
N_p	Number of processors in the system	S_A	Number of services
$R_{arr}^i(\Delta)$	Estimated packet arrival rate for service i	t_{pkt}^i	Processing time per packet for service i
t_{sw}	Context switch overhead	D^i	Delay bound for service i
G^i	Guaranteed allocation to a service i	N_{reqd}^i	Processors required by i at any instant
N_{alloc}^i	Processors allocated to i at any instant	t_{iat}^i	Minimum inter-arrival time of packets

where G^i denotes the fraction of total processors in the system that are guaranteed to be allocated to service i when the system is overloaded (i.e., the cumulative requirements of services exceed the processing capacity of the system).

Observe that, when the system is underloaded, a service may be allocated greater than $G^i \times N_p$ processors. Hence, at time t , the number of processors N_{alloc}^i allocated to service i satisfies the relationship $N_{min}^i \leq N_{alloc}^i \leq N_p$.

3. Design Considerations

To address the above two requirements, a scheduler for multi-service routers should balance *agility* and *wariness*.

1. **Agility:** Since bursts of packets can arrive at high rates ($t_{iat}^i \ll t_{sw}$), and $D^i \approx t_{sw}$, it is crucial to detect the right instants at which: (1) processors should be allocated to a service to minimize the number of packets whose delay will exceed D^i , and (2) processor should be released to provide the desired isolation among services. Delayed reaction can lead to significant number of packets suffering delays greater than D^i .
2. **Wariness:** Since $t_{pkt}^i \ll t_{sw}$, and each context switch makes a processor unavailable for t_{sw} time units, context switches reduce the effective processing capacity available for serving packets. Hence, it is crucial to switch a processor's context *only* when necessary.

In what follows, we first address the question, *are any known schedulers applicable in high-performance, multi-service routers?* We present a taxonomy of multi-processor schedulers and argue that most of the well-known schedulers fail to balance agility with wariness, and hence are not suitable. We then utilize the taxonomy to design Everest.

3.1. Multi-processor Scheduler Taxonomy

To multiplex a set of processors among services, any multi-processor scheduler addresses four questions.

1. When does a service become eligible to receive additional processors? To determine this, schedulers *monitor* the system, and decide *when to trigger* additional allocations.

2. When does a service release a processor? Services can release processors *voluntarily* or a scheduler can *force* them to release processors.
3. When a service becomes eligible, how is a victim service (from whom a processor is reclaimed) chosen? A scheduler addresses this question under two scenarios: (1) when the system contains unallocated processors (i.e., the system is *underloaded*), and (2) when all the processors are allocated (i.e., the system is *overloaded*).
4. When a service releases processors, how are beneficiary services (who receive the released processors) chosen? A scheduler addresses this question under two scenarios: (1) when the number of released processors exceeds the requirement of eligible services (*underload*), and (2) when the requirement of eligible services exceeds the number of released processors (*overload*).

We first discuss how several well-known schedulers address these questions (also see Table 4 for a summary), and then argue that these schedulers are unsuitable for high-performance, multi-service routers.

Static allocation: State-of-the-art programming environments for multi-processor, multi-service routers allocate processors to services statically [7, 28, 30]. During system operation, no service becomes eligible to receive any additional processors, and no service releases processors either voluntarily or is forced by an external event. Hence, no victim or beneficiary selection is necessary.

Quantum-based schedulers: Quantum-based schedulers [8, 11, 13, 16, 18, 23, 27, 31, 32, 37, 40] in both conventional operating systems and real-time systems allocate a processor to a task for a time quantum Q such that $Q \gg t_{sw}$. In these schedulers, a service with a non-zero queue is eligible to receive an additional processor. A service releases a processor voluntarily when its queue becomes empty, or is forced to release a processor at the end of the quantum. Beneficiaries for released processors are chosen using a wide range of policies, including round-robin or proportionate-share.

Cohort scheduler: In Cohort scheduling [22], the processor allocation eligibility and beneficiary selection are similar to quantum schedulers. However, a service releases processors

only voluntarily when its queue becomes empty. This allows Cohort scheduler to maximize data and instruction cache locality. Since processors are only released voluntarily, Cohort scheduler does not involve any victim selection.

Observation-based schedulers: In these schedulers, the eligibility for processor allocation or release is based on some function of the observed arrival rate or performance of requests. For instance, Chandra et al. [12] and Abdelzaher et al. [6] measure the average latency and arrival rate in fixed intervals, and when the latency exceeds a threshold, adjust the allocation of processors across services. SEDA [39] monitors the 90th percentile latencies suffered by requests and allocates extra resources if the latency exceeds a threshold. Services release threads (processors) to a free pool when they are idle for a certain time duration.

3.2. Limitations of Past Schedulers

To address the challenges unique to multi-service routers, a scheduler should carefully balance agility and wariness. We now discuss qualitatively why existing schedulers fail to do the same.

Static allocation: Since packet arrival rates for services fluctuate significantly, static allocation requires substantial overprovisioning of processors for each service to process packets within their delay tolerance [20].

Quantum-based schedulers: In a multi-service router, where $D^i \approx t_{sw}$, the choice of the quantum size is crucial. Small quantum sizes make the scheduler more agile but less wary; they lead to substantial reduction in processing capacity of the system because of significant number of context-switches. A large quantum size ($Q \gg t_{sw}$) makes the scheduler more wary but less agile, thereby letting packets miss their delay tolerance; packet delays can be of the order of Q . Further, since network traffic fluctuates, any single quantum size may not be the best value at all times.

Note that most real-time scheduling algorithms proposed in the literature assume that processors can be reallocated at task or request boundaries; further, they assume that the context switch overhead is negligible as compared to the task's processing time and deadline (or delay tolerance) [8, 11, 23, 31]. This assumption is not valid in a multi-service router since $t_{pkt}^i \ll t_{sw}$ and $t_{sw} \approx D^i$; hence, these algorithms may be significantly more agile and less wary than necessary and waste processing capacity in context switching. In a related effort to ours, Srinivasan et al. [32] discuss the problem of providing service guarantees to applications when allocating processing resources in programmable routers, and make similar observations as above. The authors articulate the design considerations for a scheduler for programmable routers. However, we are not aware of any follow-up work that presents the design of such a scheduler.

Cohort scheduler: Since Cohort scheduling does not force a service to release processors, the time between when a service becomes eligible and when a processor is actually allocated to the service is not bounded. This can lead to a significant number of packet experiencing delays greater than D^i , and the lack of performance isolation across services in overload.

Observation-based schedulers: Observation-based schedulers that adapt processor allocations when packets observe degraded performance (such as exceeding delay tolerance) are often less agile than necessary; since $t_{lat}^i \ll t_{sw}$ in multi-service routers, many packets that arrive after the packet that triggered processor allocation can experience delay greater than D^i before the new processor is available to the service. On the other hand, schedulers that adapt processor allocations based on predictions of packet arrival rate are often not wary; the difficulty in predicting the arrival rates accurately [26, 42] leads to unnecessary context switches.

4. Design of Everest

The design of Everest is guided by the following principles.

- Make a service eligible to receive additional processors *only when* and *as soon as* it is evident that the current processor allocation is not sufficient to serve all the packets queued at the service within their delay tolerance. When a service becomes eligible, the processor of *least utility* to other services should be selected for allocation; this maximizes the possibility that the victim service will not request reallocation of the processor in the near future.
- Make a service release processors voluntarily *only when* and *as soon as* the residual processing capacity (after releasing the processors) is sufficient to process the incoming packets within their delay tolerance. To minimize unnecessary context switches, the released processor should continue to process packets for the current service until another service is eligible to receive the processor.
- In overload (i.e., when the total demand for processors exceeds N_p), to provide performance isolation across services, the scheduler should first allocate N_{min}^i processors to each service i , and then distribute the remaining processors among services to minimize the number of packets exceeding their tolerance.

Algorithm 1 shows the pseudo-code for Everest. Lines 3 and 4 represent allocation eligibility, lines 6 and 7 represent voluntary processor release, and lines 5 and 8 represent victim and beneficiary selection respectively. Note that

Algorithm 1 Everest Algorithm

```

1: while (1) do
2:   for (each service  $i$ ) do
3:     if ( $q_{len} > Q_{lim}^j$ ) then
4:       Make service  $i$  eligible to receive processors
5:        $victim\_select(i)$ 
6:     else if ( $q_{len} = 0$ ) then
7:       Mark (or unmark) processors such that at most
8:          $k^j$  are marked
9:        $beneficiary\_select(i)$ 
10:    end if
11:  end for
12: end while

```

forced processor release is a part of victim selection. In what follows, we derive the details of allocation eligibility, processor release, beneficiary and victim selection. Since the derivation of allocation eligibility and processor release are per-service, for brevity, we will eliminate any reference to a specific service (and drop the superscript i) in the next two subsections. We will re-introduce i for beneficiary/victim selection. Also, to represent processors allocated to a service, we use the variable j interchangeably with N_{alloc}^i for brevity.

4.1. Processor Allocation

To determine the requirement for additional processors as soon as possible, Everest monitors and reacts to the state of each service's packet queue. In particular, given a delay tolerance D on a service and current level of processor allocation j , Everest determines a threshold Q_{lim}^j on each service's queue. Everest monitors the length of the packet queue for each service and declares a service eligible to receive a new processor *if and only if* the service's queue length exceeds Q_{lim}^j .

Deriving Q_{lim}^j : The queuing delay observed by a packet depends on the queue length when the packet arrives and the rate at which packets are serviced from the queue. The rate at which packets are serviced from the queue is a function of the number of processors j allocated to the service; the number of packets that can be serviced within x units of time is

$$P_{dep}^j(x) = \left\lfloor \frac{x}{t_{pkt}} \right\rfloor \times j \quad (2)$$

Let Q_{lim}^j be the maximum queue length that j processors can process within the delay tolerance D . If there are Q_{lim}^j packets in the queue and the service is allocated j processors (that are currently processing j packets), then the total number of packets in the system for the service is $Q_{lim}^j + j$. Hence, the maximum length Q_{lim}^j that a queue can grow to

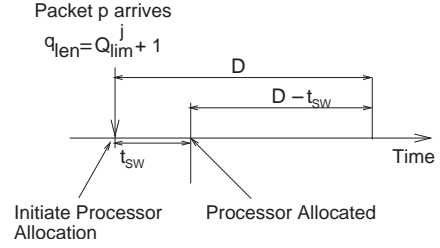


Figure 2. Processor allocation timing diagram.

without exceeding the delay tolerance for any packet is given by:

$$Q_{lim}^j + j = P_{dep}^j(D) \Rightarrow Q_{lim}^j = j * \left(\left\lfloor \frac{D}{t_{pkt}} \right\rfloor - 1 \right) \quad (3)$$

Observation: Suppose the arrival of a packet p causes the queue length to become $Q_{lim}^j + 1$ (and hence the total number of packets in the service to become $Q_{lim}^j + 1 + j$). Then, with only j allocated processors, the delay incurred by packet p would exceed D . To ensure that packet p can be serviced within its delay tolerance, an additional processor must be allocated. Observe, however, that since allocating a processor incurs an overhead of t_{sw} , packets will continue to be processed with j processors for t_{sw} duration; only after that time, $(j + 1)$ processors will begin processing packets (see Figure 2). Hence, packet p will be processed before delay D only if the total packets processed by j processors in D units of time and the $(j + 1)^{th}$ processor in $D - t_{sw}$ units of time includes p , i.e.,

$$Q_{lim}^j + 1 + j \leq P_{dep}^j(D) + P_{dep}^1(D - t_{sw}) \Rightarrow Q_{lim}^j + 1 + j \leq j * \left\lfloor \frac{D}{t_{pkt}} \right\rfloor + \left\lfloor \frac{D - t_{sw}}{t_{pkt}} \right\rfloor \quad (4)$$

By substituting Q_{lim}^j from Equation 3 and simplifying, the above requirement reduces to

$$D \geq t_{pkt} + t_{sw} \quad (5)$$

Equation 5 indicates that if $D \geq (t_{pkt} + t_{sw})$, then for all values of j the packet that triggers a new processor allocation will be serviced before its delay tolerance. When $D < (t_{pkt} + t_{sw})$, the packet that triggered processor allocation as well as some of the follow-on packets may experience delays greater than the tolerance. In this case, it is possible to adjust the queue threshold to allocate processors proactively (in anticipation of the event that queue length will exceed Q_{lim}^j); however, doing so requires prediction of the future packet arrival rate. Our experiments with such a proactive scheduler showed that significant fluctuations in packet arrival rates lead to inaccurate predictions and cause unnecessary processor reallocations; such reallocations *only increase*

the number of packets that exceed their delay tolerance when compared to Everest (For more details, please refer to [19]). To minimize such unnecessary reallocations, Everest allocates a processor *only* upon receiving a packet that results in the queue length exceeding Q_{lim}^j .

4.2. Processor Release

Voluntary release: When the queue for a service i becomes empty, Everest *marks* for release all processors that are not required to process packets at the current arrival rate. In particular, if $R_{arr}(\Delta)$ is the current arrival rate of packets for service i estimated over an interval Δ , then Everest marks k^j processors for release only if:

$$\begin{aligned} P_{dep}^{(j-k^j)}(t_{pkt}) &\geq R_{arr}(\Delta) \times t_{pkt} \\ \Rightarrow k^j &= \lfloor j - R_{arr}(\Delta) \times t_{pkt} \rfloor \end{aligned} \quad (6)$$

Processors marked for release are retained with service i until some other service becomes eligible to receive processors. If the same service becomes eligible to receive processors before the marked processors are context-switched to another service, then the necessary number of processors are unmarked without incurring a context switch (exploiting locality); line 7 of Algorithm 1 represents this behavior. These design choices make Everest wary; they reduce the number of context-switches and also make Everest non-work-conserving. Since processors are only marked for release but not switched immediately to another service (unless an eligible service exists), Everest can tolerate inaccuracies in $R_{arr}(\Delta)$ estimates.

Forced release: If a service is allocated processors beyond its guaranteed allocation, Everest can reclaim the extra processors (during victim selection) through an interrupt to satisfy the guaranteed allocation of a different service; this enables Everest to achieve the desired isolation across services.

4.3. Victim Selection

When a service requires an additional processor, a victim service is selected to reclaim a processor. At this instant, a system can be either underloaded or overloaded; the system is underloaded if there are processors marked for release, otherwise it is overloaded.

Underload: When the system is underloaded, our objective in selecting a victim service is to pick the processor that has the least *utility* at that instant. In particular, Everest selects the processor that has been marked for release for the longest duration; we refer to this as the Least Recently Allocated (LRA) policy. Using LRA reduces the chances of context switching due to transient fluctuations in traffic. Also, this policy ensures that processors remain with a service unless the service is chosen as a victim; only the processors that need to be re-allocated pay the context switch overhead.

Overload: When the system is overloaded, Everest provides the desired isolation across services using the overload control policy outlined in Section 4.5.

4.4. Beneficiary Selection

When a service i releases processors, beneficiary services are chosen to receive the released processors. At this instant, the system can be in two states, (1) the number of released processors exceeds the requirement of eligible services (*underload*) and (2) the requirement of eligible services exceeds the number of released processors (*overload*).

Underload: If the requirements of the eligible services are less than or equal to the marked processors N_{marked}^i , then all the requirements are satisfied by a subset of service i 's marked processors. The rest of the marked processors (if any) will remain allocated to service i (and process any future packets) until a service k becomes eligible and selects i as the victim service. This design decision of maximizing locality avoids unnecessary context switches.

Overload: If the requirements of the eligible services are higher than the marked processors, Everest uses the overload control policy outlined in Section 4.5.

4.5. Overload Control

When the system is overloaded, i.e. the cumulative requirements of services exceeds the number of processors in the system, Everest uses the following two-step policy to provide isolation across services.

1. Everest first ensures that each service is allocated processors equal to the minimum of the guaranteed allocation to the service and its current requirements.

$$N_{newalloc}^i = \min(N_{reqd}^i, G^i \times N_p)$$

2. Let $N_{extra} = N_p - \sum_i N_{newalloc}^i$, denote the extra processors. Our objective in distributing the extra processors is to minimize the maximum queue build-up for each service to ensure that minimum packets (1) experience delays greater than their delay tolerance, and (2) get dropped due to queue overflow. To do so, we allocate each free processor to the service that has the maximum *residual queue length*—the difference between the queue length and the current threshold on the service. This policy has a bias towards services with longer backlogged queues. When there are two services with maximum residual queue length, the service with $N_{alloc}^i > N_{newalloc}^i$ is chosen to avoid a context switch. If neither service satisfies the condition, a service is chosen arbitrarily.

Observe that when all services are overloaded and $G^i = 1$, only step 1 gets executed; in this case, any adaptive system can do only as good as a system that statically allocates processors in the ratio of G^i .

5. Evaluation

In this section, we demonstrate the effectiveness of Everest using trace-driven simulations. Our simulator is built using CSIM [15] event driven simulator.

5.1. Experimental Methodology

Table 2. t_{pkt} for IPv4-v6 gateway services.

Service	t_{pkt} (μ s)	Service	t_{pkt} (μ s)
v4→v6/TCP	18.22	v6→v4/TCP	10.09
v4→v6/UDP	13.27	v6→v4/UDP	9.72
v4→v6/ICMP	4.64	v6→v4/ICMP	1.78

Network service: We present our results in the context of the IPv4-v6 gateway application (NAT-PT) [36] that enables seamless communication between devices that only support IPv4 and devices that only support IPv6 through protocol translation. Further, depending on whether the packet type is TCP, UDP or ICMP, the application translates transport identifiers (e.g. TCP and UDP port numbers, ICMP query identifiers), and updates header checksums [36]. We profile a NAT-PT application [3] and measure the time required to process a packet at each of the six services (Table 2).

Table 3. Trace characteristics.

Trace	Peak b/w (in Gbps)	Duration (secs)	#pkts (million)	Collected on (Sept 2000)
UNC-2.5	2.5	42.4	30	29th, 11am
UNC-1	1	83.2	30	29th, 11am
UNC-10	10	21.1	60	29th, 11am
UNC-58	2.5	70.1	30	25th, 8.30am
UNC-51	2.5	59.3	30	25th, 11am

Traces: We use traffic traces collected at an edge link connecting UNC, Chapel Hill to its ISP [29]. Additionally, we derive representative traces for higher bandwidth links using well-known trace scaling techniques [10]; we scale the number of flows observed in unit time, and retain the flow level properties (see Table 3). Each UNC trace consists of two packet sequences—incoming and outgoing. To emulate the behavior of an IPv4/v6 gateway, we treat all packets on incoming link as IPv4 packets, and those on the outgoing link as IPv6 packets.

Proportional share: Through trace analysis, we determine the maximum processor requirement N_{max}^i of each service to process all the packets within their delay tolerance. The proportional share for each service i is defined as

$$G^i = \frac{N_{max}^i}{\sum_1^S N_{max}^i} \quad (7)$$

Scheduling algorithms: We compare Everest with several well-known schedulers as described below (Table 4).

Static: A system with static allocation assigns processors to services at the beginning of the experiment in proportion to the guaranteed allocation [7, 28].

Quantum: Quantum schedulers [11, 13, 16, 18, 27, 32, 37, 40] make services eligible eagerly when $q_{ten} > 0$, and release processors after a quantum expires or when a service’s queue becomes empty.

Cohort-E: Cohort scheduler [22] makes services eligible eagerly when $q_{ten} > 0$, and releases processors only after a service’s queue becomes empty. We enhance the scheduler (to Cohort-E) to proportionally allocate processors at each adaptation step; a released processor is allocated to a service such that at that instant processors are allocated in proportion to queue lengths.

Everest-LD: We define Everest-LD (represents observation-based schedulers like SEDA [39]). Unlike Everest, Everest-LD (1) adapts processor allocations to a service only *after* a packet of the service suffers delay greater than its tolerance, and (2) separates two consecutive processor allocations to a service by at least t_{sw} units of time to ensure that the previous allocation has taken effect prior to initiating a new allocation.

Everest-AE: We define a work-conserving variant of Everest, Everest-AE, whose allocation eligibility is a prioritized combination of Everest and Cohort-E. In particular, services with queue lengths exceeding Q_{lim}^j are designated as eligible to receive additional processors with a higher priority, and all services with a non-zero queue length are eligible with a lower priority. Processors are released by a service as soon as its queue becomes empty, just like Quantum or Cohort-E. However, unlike Cohort-E, the prioritized eligibility criteria allows Everest-AE to release processors by force for providing isolation.

5.2. Benefits of Everest

The experiments in this section demonstrate the benefits of balancing agility and wariness in Everest. Figure 3(a) compares the percentage of packets that suffer delays greater than the service’s tolerance with different schedulers in a system with 16 processors, $t_{sw} = 2ms$, and UNC-2.5 trace (The parameter setting $t_{sw} = 2ms$ represents the time it takes to switch the context of a processor from one service to another on the Intel’s IXP2400 platform [17]; we perform a sensitivity analysis with different values of t_{sw} in Section 5.3). For Everest and Everest-LD, the arrival rate $R_{arr}(\Delta)$ for each service is estimated as the rate of packets received in the last t_{sw} amount of time, i.e., $\Delta = t_{sw}$. For Quantum schedulers, we experiment with different quantum sizes. We only show results for two quantum sizes; the observations with other quantum sizes are similar.

Figure 3(a) shows that Everest reduces the number of packets that suffer delays greater than D by orders of magnitude compared to other schedulers. The difference between

Table 4. Taxonomy of multi-processor schedulers for comparison.

Algorithm ↓	Allocation Eligibility		Processor Release		Victim		Beneficiary	
	Monitor	Trigger	Voluntary	Forced	Underload	Overload	Underload	Overload
Static	–	$q_{len} = \infty$	–	–	–	–	–	–
Quantum	q_{len}	$q_{len} > 0$	$q_{len} = 0$	$T_{alloc} > Q$	–	Prop share	Locality	Prop share
Cohort	q_{len}	$q_{len} > 0$	$q_{len} = 0$	–	–	–	Locality	RR
Observation (e.g. SEDA)	pkt delay	90-th pkt delay $> \delta$ and $T_{lastalloc} > t_{sw}$	proc idle $> \epsilon$	–	–	–	Free pool	Prop share
Everest	q_{len}	$q_{len} > Q_{lim}^j$	Mark k^j when $q_{len} = 0$ and $R_{dep} > R_{arr}$	Interrupt	LRA	Prop share	Locality	Prop share
Everest-LD	pkt delay	pkt delay $> \delta$ and $T_{lastalloc} > t_{sw}$	Mark k^j when $q_{len} = 0$ and $R_{dep} > R_{arr}$	Interrupt	LRA	Prop share	Locality	Prop share
Cohort-E	q_{len}	$q_{len} > 0$	$q_{len} = 0$	–	–	–	Locality	Prop share
Everest-AE	q_{len}	Hi: $q_{len} > Q_{lim}^j$ Lo: $q_{len} > 0$	$q_{len} = 0$	Interrupt	LRA	Prop share	Locality	Prop share

Everest and Everest-LD demonstrates the benefit of *agility*. Since processors are allocated late with Everest-LD, queues at services build up and cause greater number of packets to exceed their delay tolerance. This phenomenon can be observed in Figure 3(b) that plots the distribution of instantaneous queue lengths of all services with various schedulers for the entire trace duration. Increased number of packets missing delay tolerance also triggers greater number of processor context switches as demonstrated by Figure 3(c).

The difference between the various work-conserving schedulers (Quantum, Cohort-E and Everest-AE) and Everest in Figure 3(a) demonstrates the benefits of *wariness*. Recall that these schedulers make services eligible eagerly when $q_{len} > 0$, and release processors as soon as a service’s queue becomes empty. This eager behavior leads to substantially larger number of context switches than necessary to process packets within their delay tolerance, thereby reducing the effective processing capacity. This leads to larger queue buildup (as demonstrated by Figure 3(b)) and results in a large number of packets suffering delays greater than the tolerance D . In contrast, Everest makes a service eligible *only* when it receives packets that will experience delay greater than their tolerance, and releases processors only after the requirement drops below the processing capacity for some amount of time. Observe that Everest-AE performs better than Quantum and Cohort-E because of the prioritized allocation eligibility (see Table 4). Unlike Quantum and Cohort-E, Everest-AE has more knowledge of when a packet may exceed its delay tolerance, and triggers processor allocation as soon as required.

Figure 3(d) shows the variation of the percentage of packets that observe delay greater than the tolerance ($D = 2ms$) in a system with different number of processors (and with $t_{sw} = 2ms$). The graph shows that Everest outperforms the

other schedulers by orders of magnitude at various levels of provisioning due to its agility and wariness.

5.3. Design Space Exploration

The design of Everest is motivated by the relationship between the constraints of state-of-the-art programmable hardware and the characteristics of network services, $D \approx t_{sw}$ and $t_{sw} \gg t_{pkt}$. In this section, we expand the design space and study the applicability of the schedulers to evolving router hardware and network services by varying the relationships between the system and application parameters D , t_{pkt} and t_{sw} . To make the experiments tractable, in this paper, we hold t_{pkt} constant and only vary D and t_{sw} . The results hold for different and widely varying values of t_{pkt} .

Figure 4(a) compares the percentage of packets that suffer delays greater than D with varying ratio of t_{sw} and D when using different schedulers. The system contains 16 processors, D is set to $500 \mu s$, the workload used is UNC-2.5 trace, and t_{sw} is varied. The graph shows that at smaller values of t_{sw} , work-conserving schedulers (Quantum, Cohort-E and Everest-AE) perform better than Everest and Everest-LD; by making all services with packets to process eligible to receive additional processors, the schedulers can maintain smaller queues at the cost of small context switch overhead. Everest-AE performs better than Quantum and Cohort-E because of the prioritized allocation eligibility (see Table 4). As t_{sw} increases, work-conserving schedulers waste significant processing capacity in eager context switching, and thereby end up with longer queues. By minimizing context switches enables Everest to minimize reduction in the effective processing capacity. Figure 4(b) shows the same observations for $D = 2 ms$.

Figure 5 compares the schedulers with respect to the average delay observed by packets in systems with different

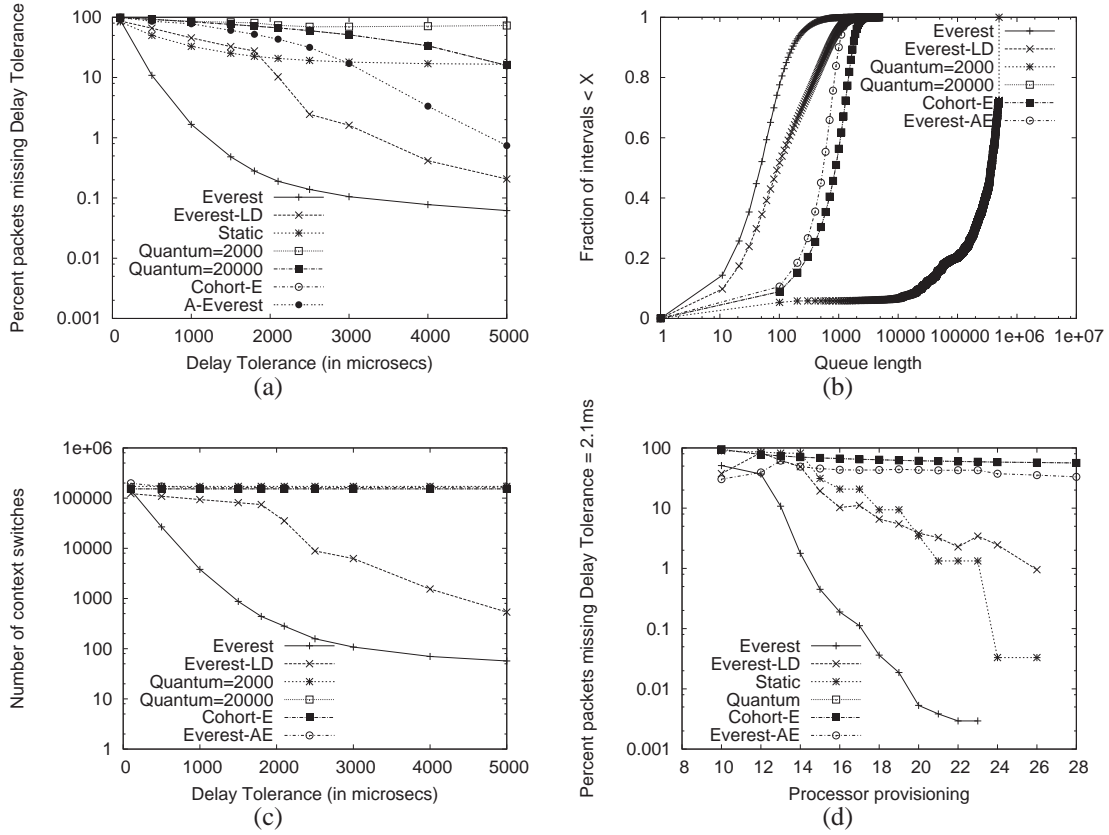


Figure 3. (a) Percentage of packets that exceed their delay tolerance (D) with increasing D . (b) Queue length CDF for $t_{sw} = 2$ ms. (c) Number of context switches with increasing D . (d) Percentage of packets whose delay exceeds D with increasing N_p .

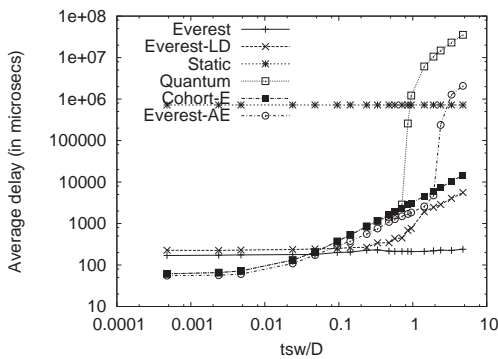


Figure 5. Average packet delay.

values of $\frac{t_{sw}}{D}$ at $D = 2$ ms. At large values of $\frac{t_{sw}}{D}$ Everest, although non-work-conserving, achieves an order of magnitude smaller average delay when compared to various work-conserving schedulers. Further, the graph demonstrates the effectiveness of Everest even in systems with much smaller context-switch times ($0.05 < \frac{t_{sw}}{D} < 1$) than state-of-the-art.

We observed the same behavior of the schedulers under several workload conditions by using traces with different characteristics as shown in Table 3(b). One key observation we made is that at various values of $\frac{t_{sw}}{D}$, one of Everest and Everest-AE outperform all other schedulers.

6. Prototype

We have implemented Everest on a platform consisting of a RadiSys ENP-2611 board, with a 600MHz IXP2400 network processor [17]. The IXP2400 processor contains one XScale controller and eight RISC cores called microengines (MEs). The XScale runs MontaVista Linux, and is used to control what runs on the MEs. Each ME has a private instruction store of size 4K instructions, on which the code for a service is loaded. All instances (loaded on different processors) of a service share the same packet queue. Everest executes as a thread on the XScale. Everest monitors each service's input packet channel to estimate the packet queue length and arrival rate, and adapts the allocation of MEs to

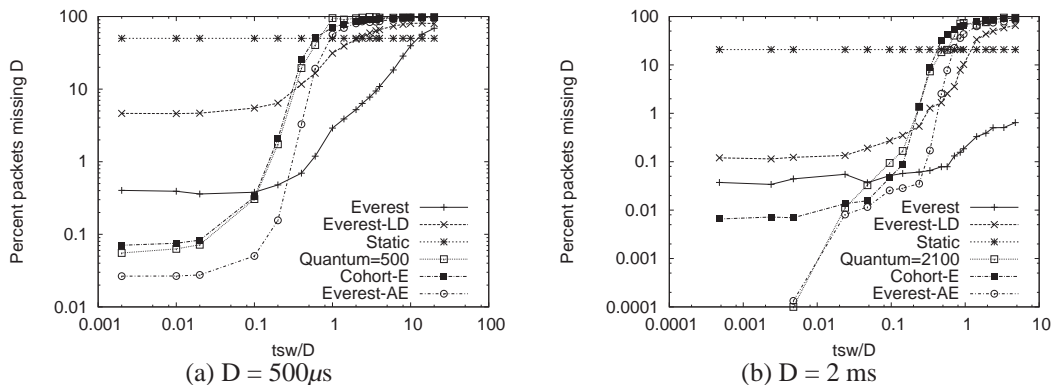


Figure 4. Behavior of schedulers with varying relationship between D and t_{sw} for various values of D .

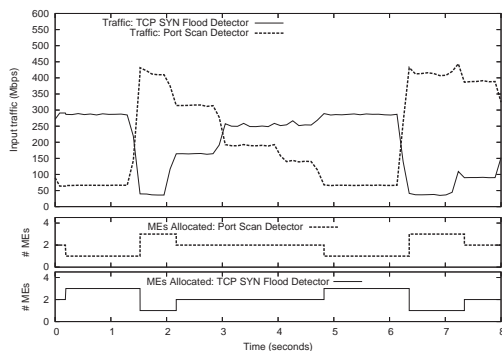


Figure 6. Adaptation of MEs among services.

services. Adapting processor allocations also requires several mechanisms to safely checkpoint services and migrate data, and utilize special hardware features (e.g., fast message passing and shared memory communication) to maximize throughput. Detailed discussion of the mechanisms and implementation is presented elsewhere [19].

To demonstrate that Everest effectively multiplexes MEs among competing services, we consider a simple deployment scenario with two services—portscan detector [41] and TCP-SYN flood detector [38]. Two MEs are used for the receive and transmit drivers that read/write packets from/to the network ports, and two MEs classify packets to be belonging to one of the two services. Hence, four MEs are available to be multiplexed between the portscan and SYN-flood detector services. We use the IXIA packet generator [2] to simulate workload fluctuations for each service (see the top graph in Figure 6). Figure 6 shows that as traffic mix changes with time, Everest adapts the allocation of MEs to match the demands of the two services. During overload (between 2.2 and 4 seconds), processors are allocated to services according to proportional share; for this experiment, we chose the share to be equal for both services.

7. Conclusion

We describe Everest, a novel processor scheduler for high performance, multi-service routers. Everest maximizes the number of packets processed within a given delay tolerance, while isolating the performance of services from each other. By incorporating two key properties—agility and wariness—Everest addresses several domain-specific challenges. (1) difficult-to-predict and high packet arrival rates, (2) small delay tolerances of packets, and (3) significant context-switch overheads.

We study the applicability of different schedulers to the evolving router hardware and network services by varying the relationships between different system and application parameters. We observe that for various ratios of t_{sw} and D , one of Everest and Everest-AE outperform all other schedulers. While it is desirable to define a single scheduler whose behavior follows the *lower envelope* of Everest and Everest-AE (refer Figure 4), the design of such a scheduler can be challenging [19]. We are exploring the design of such a scheduler as a part of our ongoing work.

References

- [1] Akamai Technologies, Inc. <http://www.akamai.com/>.
- [2] IXIA. <http://www.ixiacom.com>.
- [3] Linux-based user-space NAT-PT. <http://www.ipv6.or.kr/english/natpt-overview.htm>.
- [4] Report of NSF workshop on Overcoming Barriers to Disruptive Innovation in Networking, January 2005. www.arl.wustl.edu/netv/noBarriers_final_report.pdf.
- [5] The GENI Initiative. <http://www.nsf.gov/cise/geni/>.
- [6] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 2002.

- [7] M. Adiletta, D. Hooper, and M. Wilde. Packet Over SONET: Achieving 10 Gigabit/sec Packet Processing with IXP2800. *Intel Technology Journal*, 6(3), 2002.
- [8] J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *Real-Time Systems and Applications*, 2000.
- [9] V. Balakrishnan, R. Kokku, A. Kunze, H. Vin, and E. Johnson. Supporting run-time adaptation in packet processing systems. Technical Report TR-04-46, UT-Austin, 2004.
- [10] C. Barakat, P. Thiran, G. Iannaccone, C. Diot, and P. Owezarski. Modeling internet backbone traffic at the flow level. *IEEE Transactions on Signal processing. Special Issue in Networking*, 51(8), 2003.
- [11] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *RTAS*, 2001.
- [12] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurement. In *Int. Workshop on Quality of Service*, 2003.
- [13] S.-H. Chiang and M. K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996.
- [14] G. Chuanxiong. SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks. In *SIGCOMM*, 2001.
- [15] CSim 19 by Mesquite Software. <http://www.mesquite.com>.
- [16] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *OSDI*, 1996.
- [17] Intel IXP Family of Network Processors. <http://www.intel.com/design/network/products/npfamily/index.htm>.
- [18] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of SIGMETRICS*, 2004.
- [19] R. Kokku. *ShaRE: Run-time System for High-performance Virtualized Routers*. PhD thesis, The University of Texas at Austin, 2005.
- [20] R. Kokku, T. Riche, A. Kunze, J. Mudigonda, J. Jason, and H. Vin. A case for run-time adaptation in packet processing systems. *ACM SIGCOMM Computer Communication Review*, Jan 2004.
- [21] M. E. Kounavis, A. T. Campbell, S. T. Chou, and J. Vicente. Programming the Data Path in Network Processor-Based Routers. *Soft. Practice and Experience*, 2004.
- [22] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *USENIX Annual Technical Conference*, 2002.
- [23] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *RTAS*, 2000.
- [24] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, 2002.
- [25] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. In *HOTNETS III*, 2004.
- [26] Y. Qiao, J. Skicewicz, and P. Dinda. Multiscale Predictability of Network Traffic. Northwestern University. Technical report.
- [27] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Software-Based Router. In *ACM SIGMETRICS*, June 2001.
- [28] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *2nd Workshop on Network Processors (NP2)*, February 2003.
- [29] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP Protocol Headers Can Tell Us About the Web. In *Proceedings of ACM SIGMETRICS 2001/Performance 2001*, June 2001.
- [30] 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.
- [31] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Workshop on Parallel and Distributed Real-Time Systems*, 2003.
- [32] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *2nd Workshop on Network Processors*, 2003.
- [33] D. Stiliadis and A. Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Trans. Netw.*, 6(2):175–185, 1998.
- [34] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real Time Systems Symposium*, 1996.
- [35] L. Subramanian, I. Stoica, H. Balakrishnan, and R. H. Katz. Overqos: offering internet qos using overlays. *SIGCOMM Comput. Commun. Rev.*, 33(1):11–16, 2003.
- [36] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, 2000.
- [37] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.
- [38] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proc. INFOCOM '02*, pages 1530–1539, 2002.
- [39] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.
- [40] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, 1999.
- [41] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An effective architecture and algorithm for detecting worms with various scan techniques. In *Proceedings of the Network and Distributed System Security*, 2004.
- [42] Z. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-Time Scaling Behaviors of Internet Backbone Traffic: An Empirical Study. In *IEEE INFOCOM.*, 2003.