Adaptive Processor Allocation in Packet Processing Systems

Ravi Kokku   Upendra Shevade   Nishit Shah   Harrick M. Vin   Mike Dahlin
email: {*rkoku, upendra, nishit, vin, dahlin*} *@cs.utexas.edu*
University of Texas at Austin

Abstract: We present a delay-conscious processor allocation algorithm (PAL) for packet processing systems. PAL adapts the allocation of processors among packet processing services hosted by a system to minimize the number of packets that miss a configured delay bound. PAL accounts for processor reconfiguration overheads and copes well with the unpredictability of packet arrival patterns. A key contribution of PAL is its generality; it captures the adaptation opportunities in the system as a finite state automaton (FSA)—the methodology for constructing the FSA can be applied to a variety of application requirements and system configurations. We demonstrate through trace-driven simulations that PAL is robust to traffic fluctuations for a given processor provisioning level—it reduces the number of packets that miss the delay bound by orders of magnitude compared to a static system that allocates processors to services statically. Further, a static system requires 1.5-2 times the number of processors compared to an adaptive system that uses PAL to meet the same delay bounds.

1 Introduction

Today's packet processing systems allocate processors to packet processing services statically. In this paper, we address the question— *How to adapt processor allocations dynamically in packet processing systems?*

Background and Motivation Packet processing systems are optimized to process network packets efficiently. Today, packet processing systems support a wide-range of *header-processing applications* such as network address translation (NAT), protocol conversion (e.g., IPv4/v6 inter-operation gateway) and firewall; as well as *payload-processing applications* such as Secure Socket Layer (SSL), intrusion detection, content-based load balancing, and virus scanning.

The design of these systems is governed by two trends. First, packet processing systems are required to support high-bandwidth links (and hence, high throughput). In such environments, for most applications, the time to process a packet exceeds the inter-arrival time of packets at the system. Hence, to meet the throughput requirement, modern packet processing systems (and in particular, network processors (NP)) utilize multi-threaded, multi-processor architectures. For instance, Intel®'s IXP2800 network processor [1]—a building block used in a wide range of packet processing systems—includes on a single chip 16 RISC cores (referred to as micro-engines) and an XScale® core. Second, most packet processing systems today host multiple packet processing functions—referred to as *services*. The specific service(s) invoked for a packet depend on the packet's type (determined based on the packet header and/or payload) [14, 17].

For these systems, allocating processors to different services is a key problem. Today, this allocation is done statically (at design-time) [3, 25, 27]. Consequently, to guarantee robustness to fluctuations in the arrival rate for different services, systems are required to provision sufficient number of processors to handle the expected *maximum* load for each service. However, as illustrated by Figure 1(a), the observed load fluctuates significantly over time and at any instant is often substantially lower than the maximum load [18, 23, 34].

In such settings, an adaptive run-time environment—that can change the allocation of processors to services at run-time—can yield significant benefits as our results in Figure 1(b) illustrate. First, matching processor allocations to the processing demands for each service leads to system designs that are *robust* to traffic fluctuations; an adaptive system can allocate appropriate number of processors to each service even when the processing demands for a service exceed design-time expectations, as long as the cumulative processor demands across all services do not exceed the provisioning level. Second, by multiplexing processors among services, an adaptive system can reduce the cumulative processor provisioning level and system cost. Finally, by reducing the power consumed by idle processors (e.g., by turning off processors), an adaptive system can conserve energy.

Realizing the benefits of adaptation requires the system to adapt at various time-scales. While adapting at coarse time-scale in response to time-of-day variations of traffic provides most of the energy benefits [10], achieving robustness and reducing processor provisioning requires the system to adapt at fine time-scales. In this paper, we focus on multiplexing processors among services at fine time-scales, and hence the problem of maximizing the robustness and provisioning benefits in a packet processing system.

The Challenges Adapting at fine time-scales is challenging in a packet processing system because of application requirements, workload characteristics, and hardware constraints. Packet processing applications typically require packets to meet a stringent delay bound, generally of the order of hundreds of microseconds to a few milliseconds. At these time-scales, however, network traffic fluctuates significantly [23, 34]; hence, predicting traffic arrival patterns accurately to determine the processor allocation requirement is difficult. Further, allocating and releasing processors in-

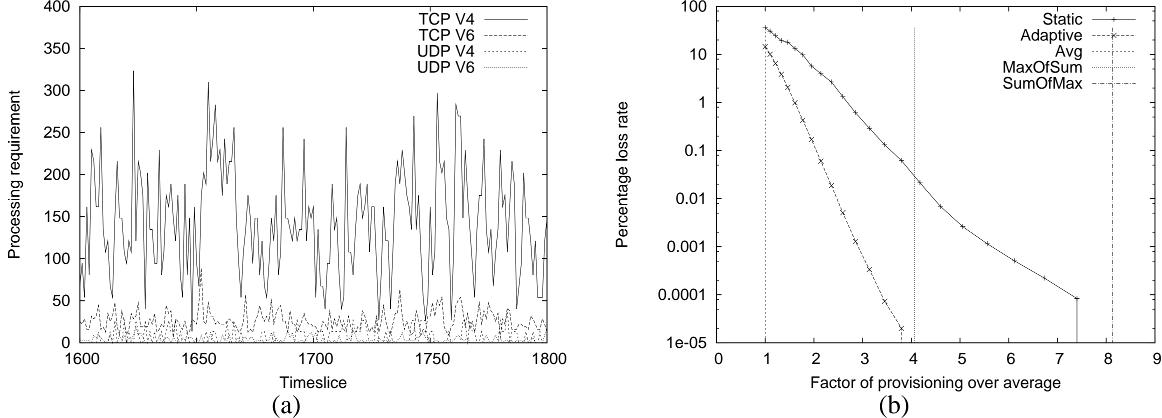

Figure 1: Graph (a) illustrates the variation in the processing requirements observed in 200 consecutive intervals (of length 1ms) of a UNC trace [26] for different services of the IPv4-v6 gateway [30]. Graph (b) illustrates the benefits of adapting processors (assuming no adaptation overhead) among the different services of the IPv4-v6 gateway. For a given level of provisioning, an adaptive system causes orders of magnitude lower packet loss rate than a statically allocated system; a packet is considered lost if it can't be processed within a certain interval of time. Conversely, to bound the loss rate to a specified amount (consider a horizontal line), a static system requires double the number of processors compared to an adaptive system. The lines MaxOfSum and SumOfMax represent the maximum provisioning required by an adaptive system and a static system respectively, to achieve zero loss rate. The experiment was conducted using a one-hour packet trace collected from the network link connecting the University of North Carolina at Chapel Hill to its ISP.

curs a delay, generally of a few hundred microseconds [19], which is similar in magnitude as the per-packet delay bound requirement. The inability to allocate an appropriate number of processors to services can cause a burst of arriving packets to suffer unacceptable delays or losses.

The problem of adapting the allocation of processors dynamically at fine time-scales has been explored in various domains. These solutions can be categorized into *request-level scheduling* [8, 29, 33] and *quantum-based scheduling* [7, 13, 15, 24, 31] algorithms. Request-level scheduling algorithms may switch a processor from providing one service to another on every request. In the case of packet processing systems, the time t_{pkt} to service a packet is often significantly smaller than the time t_{sw} to switch the context of the processor from one service to another. Hence, such request-level scheduling algorithms can waste significant resources in transitioning processors from one service to another. Quantum-based (or window-based) scheduling algorithms address the limitation of request-level schedulers by allowing a service to process a *fixed* number of requests (or for a fixed time quantum Q, $Q \gg t_{sw}$) prior to switching the processor allocation to a different service. These algorithms amortize the context-switch overhead across multiple requests; however, they can not provide delay bound smaller than the time quantum Q. For packet processing systems, the delay bound D is similar in magnitude as t_{sw}; hence, fixed quantum-based algorithms are not well-suited for this environment.

Our Contributions In this paper, we design a novel adaptive processor allocation algorithm (PAL) that allows a packet processing system to meet delay bounds for each packet while minimizing the impact of the context-switch overhead t_{sw}. PAL exploits the existence of multiple processors in packet processing systems to support the notion of *variable-size quantum*. PAL is light-weight; it does not maintain individual deadline values for each packet and imposes little run-time overhead for its execution. The design of PAL incorporates three key design decisions: (1) *early detection* – that allows PAL to determine upon a packet arrival whether the current level of processor allocation is sufficient to process the packet within its delay bound; (2) *proactive allocation* – that allows PAL to allocate an appropriate number of processors to each service to ensure that delay bound can be met; and (3) *reactive release* – that allows PAL to release processors allocated to a service only when doing so does not impact the delay bound of any packets waiting for that service.

There are two salient features of PAL. First, PAL is general. It captures the adaptation opportunities in the system as a finite state automaton (FSA). It parameterizes the different factors that impact adaptation, and determines the state transitions in the FSA based on these parameters. Hence, PAL defines a general framework that can be applied to a variety of application requirements and system configurations. Second, PAL offers the flexibility to instantiate various policies for *eager* or *lazy* allocation and release of processors; this allows PAL to trade off adaptation frequency/benefits with the delay incurred by packets.

We demonstrate the effectiveness of PAL in multiplexing processors among services of a packet processing system through analysis and simulations. We evaluate the benefits of adaptation in a system that uses PAL for a set of trace workloads, a realistic application model, and realistic processor allocation overheads. Our results show that,

for a given amount of provisioning, an adaptive system using PAL reduces the number of packets missing their delay bounds by several orders of magnitude when compared to a statically provisioned system. Further, a static system uses 1.5-2 times the number of processors required by an adaptive system to meet the same delay bounds.

The rest of the paper is organized as follows. In Section 2, we describe our system model. In Section 3, we discuss our processor allocation algorithm. We describe the results of our experimental evaluation in Section 4 . Related work is discussed in Section 5, and finally, Section 6 summarizes our contributions.

2 System Model

We consider a packet processing system with $\mathcal{P}$ processors that co-host $\mathcal{S}$ services. Each service may refer to a component [17] or a pipeline stage [32] for a packet processing application, or a new router capability deployed in virtual routers [12, 16]. At any instant, a subset of the processors in the system are allocated to each service [1]. A packet arriving into the system is processed at one or more services prior to departure. Each service is associated with a queue; packets are placed in the queue until a processor becomes available to process the packet (see Figure 2). We assume that a packet queued at service i can be served by any of the processors allocated to the service i. Let the time taken to process a packet by service i be t^i_{pkt} 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 (once the packet has been processed by all the required services).

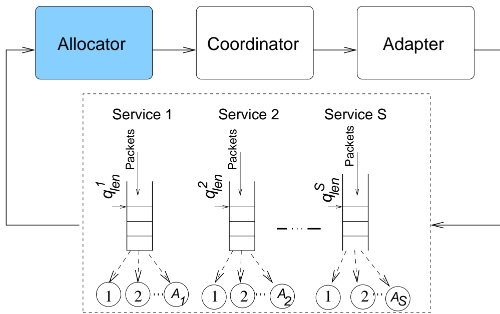

Figure 2: System model. $\mathcal{A}_i, i \in [1,\mathcal{S}]$ denote the number of processors allocated to service i.

Let the delay between when a packet is enqueued and when its processing is complete by service i be bounded by D^i. In this paper, we only model the delay incurred by packets while waiting for service by one of the processors; we do not model delay incurred by packets in the input or output ports. Let R^i_{arr} denote the maximum rate of packet arrivals into the queue for service i. Finally, let t_{sw} denote the delay incurred to switch the context from one service to another. Table 1 summarizes these system model parameters.

Parameter	Description
$\mathcal{P}$	Number of processors in the system
$\mathcal{S}$	Number of services
R^i_{arr}	Maximum packet arrival rate for service i
t^i_{pkt}	Processing time per packet for service i
t_{sw}	Context switch overhead
D^i	Delay bound for service i

Table 1: System model parameters

Our goal is to design an adaptive environment that multiplexes processors among services in a packet processing system. The environment consists of three components—an *allocator*, a *coordinator*, and an *adapter* (Figure 2). The allocator determines the processor allocation requirement for each service. The coordinator collects allocation requests and determines the assignment of processors to services according to the requirements of each service and an overload control policy. The adapter controls the transitioning of the system from the current processor assignment to the next assignment. In this paper, we focus on the processor allocation algorithm (PAL) used by the allocator. We briefly describe the functionality of the coordinator in section 4.2.2.

3 Processor Allocation Algorithm

3.1 Design Overview

The design of PAL is governed by two requirements—(1) process packets prior to their delay bounds; and (2) minimize the overhead of adaptation (or be *light-weight*). To be light-weight, PAL should minimize the amount of state maintained for adaptation, incur little per-packet processing overhead, and minimize the number of adaptations (or context switches). To meet the two requirements, PAL should allocate/release the right number of processors at the right instant to closely match the arrival rate of packets; allocating/releasing processors too frequently can increase the number of adaptations required, while allocating/releasing too infrequently may cause packets to miss deadlines. Hence, the design of PAL is guided by the following decisions.

- *Early detection*: PAL determines the processor requirement to meet the delay bound of each packet at the time of its arrival. Detecting the requirement as early as possible enables processing packets prior to their delay bounds. To do so, PAL monitors the queue length of packets at each service; when the queue length exceeds a threshold computed based on the number of processors allocated to the service, PAL adapts processor allocations.

- *Pro-active allocation*: When the currently allocated processors for a service are insufficient to process the

[1] For simplicity, we assume that at any instant a processor is allocated to at most one service. Our algorithm can be easily generalized to the case where a processor is allocated to multiple services by appropriately scaling the throughput that each service receives from the processor.

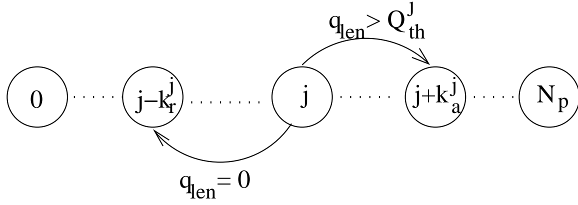

Figure 3: The finite-state automaton for PAL. The quantities in each state denote the current processor allocation.

incoming packet before its deadline, PAL allocates new processors such that the current packet, and all the packets that may arrive during the adaptation delay t_{sw}, meet their deadlines. Since predicting packet arrival rates at the adaptation time-scales is challenging [23, 34], PAL uses R^i_{arr}—an estimate of the maximum rate of packet arrivals for service i—to derive the number of packets that may arrive in time t_{sw} for service i. Such pro-active allocation of processors separates two subsequent allocation requests for a service by at least t_{sw} (i.e. a new allocation starts only after the previous allocation is complete), thereby making PAL light-weight. This is because, if allocations were separated by less than t_{sw} time units, the queue thresholds would depend on the time when the previous allocations finish, thereby requiring extra computation per packet.

During low levels of processor utilization, PAL maintains a certain level of processor allocation for each service to ensure that a burst of packet arrivals at any instant can be served prior to their delay bounds.

- *Reactive release*: PAL releases processors when the queue becomes empty. This design decision allows PAL to be light-weight. Pro-active release of processors (before the queue becomes empty) would require the algorithm to re-evaluate the possibility of delay bound violations for each packet in the queue–but doing so requires the maintenance of arrival times with each packet and imposes significant computational overhead. Further, when the queue becomes empty, PAL releases only processors that not needed to process packets at the current arrival rate. This design choice minimizes the number and the overhead of adaptations.

Based on these design decisions, PAL constructs for each service a *finite state automaton (FSA)* (see Figure 3). Each state in the FSA for service i represents a processor allocation level for service i; state transitions denote processor allocation and release events. State transitions are triggered based on the length q^i_{len} of the queue for service i.

When service i is allocated j processors, PAL requests allocation of an additional k^j_a processors (by making a transition from state j to $j+k^j_a$ in the FSA) when the queue length for service i exceeds a threshold Q^j_{th}. Similarly, when the queue is empty ($q^i_{len}=0$), then from any state j, PAL can release $k^j_r \leq (j-n_{min})$ processors, where n_{min} is the minimum number of processors that must remain allocated to service i. In what follows, we describe construction of the FSA by deriving the values of Q^j_{th} and k^j_a for all values of j, and the values of n_{min} and k^j_r. Since the construction is the same for all services, we present the FSA construction for a single service. Further, for brevity, we will eliminate any reference to a specific service (and drop superscript i from all symbols defined in Table 1) from our discussion.

3.2 Processor Allocation

3.2.1 Q^j_{th}: When to Allocate Processors?

PAL allocates one or more processors when the queue length reaches a level where the delay incurred by packets can exceed the desired bound D. 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 allocated to the service. In particular, since each processor can service a packet in t_{pkt} time, the service rate R^j_{dep} for j processors is given by:

$$R^j_{dep} = \frac{j}{t_{pkt}} \tag{1}$$

Let Q^j_{lim} denote the maximum queue length that j processors can process within the maximum permitted packet-processing delay D. Note that if there are Q^j_{lim} packets in the queue and the service is allocated j processors, then the total number of packets in the service is $Q^j_{lim}+j$. Hence, Q^j_{lim} is determined by

$$Q^j_{lim} + j = D \times R^j_{dep}$$
$$\Rightarrow Q^j_{lim} = j * \left(\frac{D}{t_{pkt}} - 1\right) \tag{2}$$

Suppose the arrival of a packet p causes the queue length to become $Q^j_{lim}+1$ (and hence the total number of packets in the service to become $Q^j_{lim}+1+j$). Then, with only j allocated processors, the delay incurred by packet p would exceed D. To ensure that packet p can be serviced prior to its delay bound, one or more additional processors must be allocated.

A newly allocated processor, however, can serve packets only after t_{sw} time units. Hence, in order to understand how the packet p meets its delay bound, let us assume that the system requests allocation of the new processor τ units of time *prior* to the arrival of packet p (see Figure 4). When $\tau > 0$, PAL *speculatively* allocates additional processors in anticipation of q_{len} exceeding Q^j_{lim}; in contrast, when $\tau \leq 0$ PAL allocates additional processors only after observing that $q_{len} > Q^j_{lim}$.

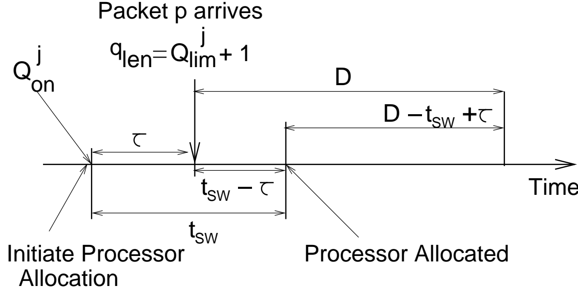

Figure 4: Allocation procedure: Timing diagram

Observe from Figure 4 that for an interval $(t_{sw} - \tau)$ after the arrival of packet p, packets are serviced with j processors (and hence, at the rate of j/t_{pkt}); after that time, $(j+1)$ processors service packets at the rate of $(j+1)/t_{pkt}$. Thus, packet p will meet its delay bound D if and only if

$$Q^j_{lim} + 1 + j \quad \leq \quad (t_{sw} - \tau) \times \frac{j}{t_{pkt}} + (D - t_{sw} + \tau) \times \frac{j+1}{t_{pkt}}$$

By substituting Q^j_{lim} from Equation 2 and simplifying, the above requirement reduces to

$$\tau \quad \geq \quad t_{pkt} + t_{sw} - D \tag{3}$$

Equation 3 leads us to the following conclusion.

Conclusion 1 *For $D \geq t_{sw} + t_{pkt}$, τ can be smaller than or equal to 0. Hence, allocation of additional processors need to be triggered only $-\tau$ units of time after the queue length q_{len} exceeds Q^j_{lim}, where j is the number of currently allocated processors. Hence, $Q^j_{th} = Q^j_{lim}$. Otherwise, if $D < t_{sw} + t_{pkt}$, then PAL must speculate the possibility of q_{len} exceeding Q^j_{lim} and thereby trigger the allocation of additional processors when $q_{len} > Q^j_{lim} - \tau \times (R_{arr} - R^j_{dep})$. Hence, if $(t_{pkt} + t_{sw} - D) > 0$, then $Q^j_{th} = Q^j_{lim} - \tau \times (R_{arr} - R^j_{dep})$.*

This is an important conclusion; it indicates that when the delay bound $D \geq t_{pkt} + t_{sw}$, PAL can observe the queue build-up and react (and incur the context-switch overhead of adapting a processor) only upon receiving a packet whose delay bound will be violated.

3.2.2 k^j_a: How Many Processors to Allocate?

During every new allocation, PAL pro-actively allocates processors to ensure that two subsequent allocations for a stage are separated by at least t_{sw} duration. Once requested, a processor becomes available to service packets only after a delay of t_{sw} time units. Thus, the number of processors to be allocated is selected such that all the packets that can arrive within time t_{sw} (not just packet p that triggered the allocation request) can be serviced prior to their respective deadlines.

If j and k^j_a, respectively, denote the number of currently allocated processors and the ones being requested, then the above condition can be met if the queue length at time when $(j + k^j_a)$ processors are ready to serve packets does not exceed $Q^{j+k^j_a}_{th} + 1$. Note that the request for additional k^j_a processors is triggered when $q_{len} = Q^j_{th} + 1$. During time interval t_{sw}, packets can arrive into the service queue at a rate R_{arr}; further, with j allocated processors, packet depart the queue at rate R^j_{dep}. Thus, the maximum increase in queue length is bounded by $(R_{arr} - R^j_{dep}) \times t_{sw}$. Thus, the delay bound for each packet can be satisfied if:

$$(Q^{j+k^j_a}_{th} + 1) - (Q^j_{th} + 1) \quad \geq \quad \left(R_{arr} - R^j_{dep}\right) \times t_{sw}$$

Substituting the values for $Q^{j+k^j_a}_{th}$, Q^j_{th}, and R^j_{dep} from Equation (1) and Conclusion (1), we get:

$$k^j_a \times \left(\frac{D}{t_{pkt}} - 1\right) + \tau \times \left(\frac{k^j_a}{t_{pkt}}\right) \geq \quad \left(R_{arr} - \frac{j}{t_{pkt}}\right) \times t_{sw}$$

$$\Rightarrow k^j_a \geq \frac{(R_{arr} \times t_{pkt} - j) \times t_{sw}}{D + \tau - t_{pkt}} \tag{4}$$

Equation 4 leads to the following conclusion.

Conclusion 2 *When the queue length for a service with j allocated processors reaches its threshold (as defined in Conclusion 1), the smallest number of processors k^j_a that should be allocated is given by:*

$$k^j_a = \min\left(\left\lceil \frac{(R_{arr} \times t_{pkt} - j) \times t_{sw}}{D + \tau - t_{pkt}} \right\rceil,\ \mathcal{P} - j\right) \tag{5}$$

where $\mathcal{P}$ is the total number of processors in the system.

We make the following observations.

- The value of k^j_a shown in Conclusion 2 is a function of j ($j \in [0, \mathcal{P}]$), the number of currently allocated processors. The smaller the value of j, the greater is the value of k^j_a, and vice versa. This relationship allows the system to ramp-up quickly from a low-utilization state (with very few allocated processors) by allocating a larger number of processors first. The number of processors allocated with each trigger decreases at higher levels of utilization. In the limit, when $j = \mathcal{P}$, no additional processors can be allocated; hence, $k^{\mathcal{P}}_a = 0$.

- Equation (3) defines a lower-bound on the value of τ (in particular, $\tau \geq t_{pkt} + t_{sw} - D$). If τ is selected to be equal to the lower-bound (i.e., $\tau = t_{pkt} + t_{sw} - D$), then Equation (5) reduces to:

  $$k^j_a = \min\left(R_{arr} \times t_{pkt} - j,\ \mathcal{P} - j\right) \tag{6}$$

  As a result, independent of the value of D, the FSA contains state transitions as determined by Equation 6, which can be large depending on the value of R_{arr}.

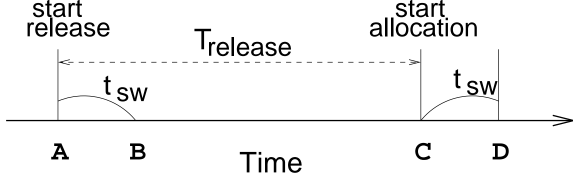

Figure 5: Condition for release to be beneficial

Selecting $\tau > t_{pkt} + t_{sw} - D$ results in smaller values of k_a^j. The greater the value of τ, the smaller the value of k_a^j. Thus, k_a^j values for a speculative system ($\tau > 0$) are smaller than those for a reactive system ($\tau \leq 0$); further, even for a reactive system, selecting $\tau < 0$ yields larger values of k_a^j as compared the case when $\tau = 0$. Smaller values of k_a^j are preferable; they allow PAL to gradually increase processor allocations and thereby achieve higher multiplexing benefits.

- If $D > (R_{arr} \times t_{pkt} \times t_{sw} + t_{pkt}) - \tau$, then from Equation (5), for all values of $j < \mathcal{P}$, $k_a^j = 1$. Thus, for a large value of D, PAL adds only one processor every time the queue length crosses a threshold, and still meets delay bounds of packets. Also, observe that the number of reachable states in the FSA becomes equal to $\mathcal{P}$. The greater the number of reachable states in the FSA, the better is the opportunity for PAL to align processor allocation to processing demand.

3.3 Processor Release

3.3.1 When to Release Processors?

As argued in Section 3.1, PAL releases processors reactively; it releases an appropriate number of processors after the queue becomes empty. Observe that the queue becomes empty when the packet processing capacity allocated to a service (and hence the maximum packet service rate) exceeds the packet arrival rate for the service.

3.3.2 k_r^j: How Many Processors to Release?

Once the queue becomes empty, processors should be released such that packets arriving in the future can still meet their delay bounds. To determine the number of processors that can be released, we first derive the minimum number of processors n_{min} that must remain allocated to ensure that the delay bound for any future packets is not violated.

To derive the value of n_{min}, we observe that releasing a processor is beneficial only if a subsequent allocation of processors to the service is separated from the release event by at least t_{sw} (the time required to release/allocate a processor)(Figure 5). Observe that any more processors than n_{min} would need to be allocated only τ units of time prior to the instant when the queue length reaches $Q_{lim}^{n_{min}} + 1$ (from its current value of 0). Given that packets can arrive at the rate R_{arr} and that packets are serviced at rate $R_{dep}^{n_{min}}$ with n_{min} allocated processors, the earliest time at which additional processors may need to be added is given by:

$$T_{release} = \frac{Q_{lim}^{n_{min}} + 1}{R_{arr} - R_{dep}^{n_{min}}} - \tau \qquad (7)$$

By requiring the $T_{release} \geq t_{sw}$, we derive n_{min} as:

$$\forall j : n_{min} \geq \frac{((t_{sw} + \tau) \times R_{arr} - 1) \times t_{pkt}}{D - t_{pkt} + t_{sw} + \tau} \qquad (8)$$

Equation 8 leads to the following conclusion.

Conclusion 3 *Once the queue becomes empty, an adaptive system can release all but n_{min} processors and still ensure that the delay bound is met for all packets. Hence, the number of processors that can be released from state j is bounded by:*

$$k_r^j \leq j - n_{min} \qquad (9)$$

We make the following observations.

- Substituting $\tau = t_{pkt} + t_{sw} - D$ in Equation (8), we get:

$$n_{min} \geq \frac{((t_{pkt} + 2 \times t_{sw} - D) \times R_{arr} - 1) \times t_{pkt}}{2 \times t_{sw}} \qquad (10)$$

  Thus, the greater the delay bound D, the smaller the value of n_{min}. In fact, if $D \geq t_{pkt} + 2 \times t_{sw}$, then $n_{min} = 0$.

- If $\tau \leq -t_{sw}$, then the condition

$$T_{release} = \frac{Q_{lim}^{n_{min}} + 1}{R_{arr} - R_{dep}^{n_{min}}} - \tau \geq t_{sw}$$

  is satisfied for all values of n_{min}, including $n_{min} = 0$.

We summarize these observations in the following conclusion.

Conclusion 4 *If $D \geq t_{pkt} + 2 \times t_{sw}$, then by selecting $\tau \leq -t_{sw}$, one can design an adaptive system in which once the queue becomes empty, the system can release all the idle processors, while ensuring that the delay incurred by each packet is bounded by D.*

3.4 Discussion

Equation (3) defines a lower-bound on the value of τ, and Conclusion (9) defines an upper-bound on k_r^j, the number of processors that can be released from state j. The design of PAL supports flexibility in selecting τ and a method for computing k_r^j. In this section, we discuss the impact of selecting different values of τ and k_r^j on the efficacy of PAL.

3.4.1 Selecting τ

From Equation (3), $\tau \geq t_{pkt} + t_{sw} - D$. Thus, if $D < t_{pkt} + t_{sw}$, then $\tau > 0$; hence, PAL operates in the speculative mode (making worst-case estimates about packet arrival rate). In this case, selecting the smallest possible value of τ (equal to $t_{pkt} + t_{sw} - D$) is the most appropriate.

In contrast, if $D \geq t_{pkt} + t_{sw}$, then $t_{pkt} + t_{sw} - D \leq 0$. Hence, PAL can allocate additional processors only after receiving a packet whose deadline will be violated with the current set of allocated processors. This case also offers flexibility in selecting a value of τ in the range $(t_{pkt} + t_{sw} - D) \leq \tau \leq 0$.

- An *eager* processor allocation policy can select $\tau = 0$. Such a policy yields smaller values of k_a^j (Equation (5)); this allows PAL to better align processor allocations with the processing demands. On the contrary, selecting $\tau = 0$ ensure that $n_{min} > 0$; hence, PAL must maintain a minimum allocation of n_{min} processors to the service even if the processing demand is smaller.

- A *lazy* processor allocation policy can select any value of τ in the range $(t_{pkt} + t_{sw} - D) \leq \tau < 0$. The extreme case is one with $\tau = t_{pkt} + t_{sw} - D$. In this case, if $D \geq t_{pkt} + 2 \times t_{sw}$, then as per Conclusion 4, $n_{min} = 0$. However, as per Equation (6), $\forall j : k_a^j = \mathcal{P} - j$. Since $n_{min} = 0$, this results in a two-state FSA – the two states represent allocations of 0 and $\mathcal{P}$ processors to the service. As a result, PAL makes frequent transitions between these two states; this minimizes processor multiplexing opportunities (and hence is not desirable).

  A more desirable policy is one that allows $n_{min} = 0$, and yet has smaller values of k_a^j. For $D > t_{pkt} + 2 \times t_{sw}$, this can be achieved by selecting $\tau = -t_{sw}$.

3.4.2 Selecting k_r^j

From Equation (9), $0 \leq k_r^j \leq j - n_{min}$. The choice of k_r^j defines the following two policies.

- An *eager* policy for releasing processors selects $k_r^j = j - n_{min}$. In particular, when $q_{len} = 0$, such a policy releases all but n_{min} processors. This policy is the most aggressive in terms of releasing idle processors; this facilitates statistical multiplexing of processors among services. However, it also introduces a significant number of allocation/release transitions in the system, thereby reducing the efficiency of the system.

- A *lazy* policy for releasing processors releases processors progressively so as to arrive at the right level of processor allocation for each service. A lazy scheme can employ various sub-policies for selecting the value of k_r^j. Some simple instances of such sub-policies include decreasing the allocation by a constant (*additive-decrease*) or by a constant factor (*multiplicative-decrease*).

  A little more involved policy is one that measures the current arrival rate of packets and releases all but the processors needed to match the arrival rate. In particular, such a policy measures over a certain time interval Δ the arrival rate $\widehat{R}_{arr}(\Delta)$ of packets, and then computes k_r^j as follows:

$$k_r^j = \min\left(j - n_{min},\ \ j - \left\lceil \frac{\widehat{R}_{arr}(\Delta)}{1/t_{pkt}} \right\rceil\right) \qquad (11)$$

  where $\left\lceil \frac{\widehat{R}_{arr}(\Delta)}{1/t_{pkt}} \right\rceil = \left\lceil \widehat{R}_{arr}(\Delta) \times t_{pkt} \right\rceil$ denotes the number of processors required to match the packet arrival rate at a service. With such a policy, the action of releasing processors is triggered when the queue is empty (i.e., $q_{len} = 0$) and the number of required processors reduces.

Observe that the *lazy* release policies introduce several reachable intermediate states in the FSA. This allows PAL to better match processor allocations to the requirements; further, it reduces the number of allocate/release transitions performed by the system.

4 Experimental Evaluation

In this section, we demonstrate the effectiveness of PAL in multiplexing processors among competing services using trace-driven simulations. In what follows, we first describe our experimental methodology, and then present the results.

4.1 Experimental Methodology

Application: We conduct our experiments with several applications. For brevity, we discuss our results in the context of one canonical packet processing application— an IPv4-v6 gateway (NAT-PT) [11, 30] that interfaces IPv6 only devices with IPv4 only devices by translating IPv6 packets to IPv4 packets and vice versa. 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 the header checksums [11, 30]. For each of the six services (See Table 2) in the application, we profile an open implementation of NAT-PT and measure the computation time required to process a packet at the service.

Service	t_{pkt} (in μs)	Service	t_{pkt} (in μs)
v4/TCP	18.22	v6/TCP	10.09
v4/UDP	13.27	v6/UDP	9.72
v4/ICMP	4.64	v6/ICMP	1.78

Table 2: IPv4-IPv6 gateway work model derived using the Performance Counters Library [6] on a 930MHz, Intel® Pentium® III system.

Trace: We drive our simulations using several traces collected from various points in the Internet. In this section, we discuss our results using traces collected from an edge link connecting University of North Carolina at Chapel Hill (UNC) to its network service provider [26]. Each UNC trace contains 30 million packets and is about 10 minutes long. The trace consists of two packet sequences—incoming and outgoing. To emulate the behavior of a typical IPv4/v6 gateway, we consider all incoming and outgoing packets as IPv4 and IPv6 respectively, and the application converts them into IPv6 and IPv4 respectively. We use well-known techniques [5] to scale the UNC traces (of peak utilization 100Mbps) to emulate an OC-48 link of peak utilization 2.5Gbps.

For each service, we analyze the trace to determine the maximum arrival rate of packets—the maximum number of packets that arrive in a moving window of size t_{sw}—and set R_{arr} to the maximum rate.

Policies: PAL provides the flexibility of defining various allocation and release policies. We consider the following allocation and release policies.

- We consider two allocation policies: (1) **Eager-Alloc** with $\tau = \max(t_{pkt} + t_{sw} - D, 0)$; and (2) **Lazy-Alloc** with $\tau = \max(t_{pkt} + t_{sw} - D, -t_{sw})$.
- We consider two release policies: (1) **Eager-Rel** with $k_r^j = (j - n_{min})$; and (2) **Lazy-Rel** with $k_r^j = \min\left(j - n_{min},\ j - \left\lceil \frac{\widehat{R}_{arr}(\Delta)}{1/t_{pkt}} \right\rceil\right)$ (see Equation 11).

For all the experiments, we assume that all services provide the same delay bound to packets. Unless specified, we use t_{sw} = 200μs to re-allocate a processor from one service to another.

4.2 Results

We present our results in three sets. The first set of experiments explores the behavior of PAL in detail— (1) it shows the dynamics of PAL, (2) it demonstrates the flexibility of PAL in instantiating various policies, and highlights the properties of various policy combinations, and (3) it compares the provisioning requirements of PAL to various systems. The second set of experiments demonstrates the robustness and provisioning benefits of using PAL in a realistic system setup. The third set analyzes the sensitivity of the benefits of adaptation to t_{sw}.

4.2.1 Algorithm properties

Algorithm Dynamics: Figure 6 shows the dynamics of PAL for one service (v6/TCP) with a synthetic trace (used only for this experiment) that is designed to highlight PAL's proactive allocation and reactive release of processors. For this experiment, we use Lazy-Alloc-Lazy-Rel policy combination. The graph plots with time, the number of packets for the v6/TCP service in the trace in each millisecond interval, the queue length seen by the service, and the number of processors allocated by PAL. The graph demonstrates that PAL

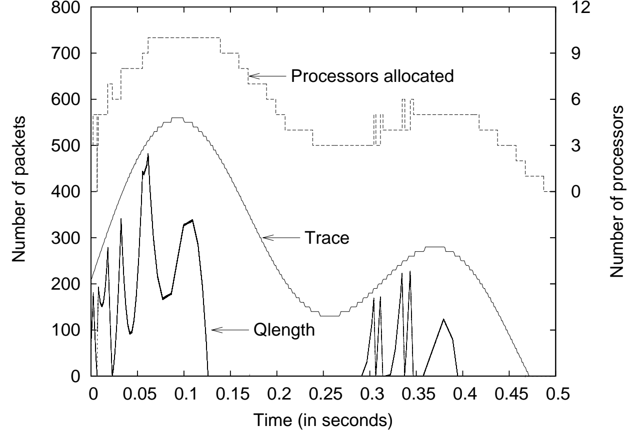

Figure 6: Dynamics of PAL.

closely follows the trace by allocating processors whenever the queue length crosses a threshold, and by releasing processors as the current arrival rate of packets decreases. The graph also demonstrates that PAL tries to ramp-up quickly by adding more processors when the currently allocated processors are few. When the number of processors are already high, PAL adds fewer processors at a time. When the queue length becomes zero, PAL with Lazy-Rel releases processors based on the current arrival rate.

FSA structure: Figure 7 shows the FSAs for the v6/TCP service for different values of D and different combinations of PAL policies. For each value of D, the number in the circle denotes the number of processors currently allocated to the service, left most circle represents n_{min}, and the arrows represent the transitions from the state j, when k_a^j processors are added. The length of the transitions represents the value of k_a^j. For the v6/TCP service, $R_{arr} = 10^6$ packets per second.

We make multiple observations from Figure 7. First, as D increases, both n_{min}, and k_a^j for any value of j (i.e., the length of the transitions) decrease. Second, n_{min} decreases more slowly with D for Eager-Alloc, while it quickly reaches zero with Lazy-Alloc. Third, since $\tau_{Lazy-Alloc} \leq \tau_{Eager-Alloc}$ for all values of D, k_a^j (for all values of j) for Eager-Alloc is never higher than Lazy-Alloc for a given value of D. Finally, k_a^j reduces with increase in j. Observe that the release policies control the number of reachable states in the FSA. When the algorithm uses Eager-Rel, since $k_r^j = (j - n_{min})$, the set of reachable states consists of only the states that can be reached by a sequence of increases in processor allocations starting from state n_{min}. With Lazy-Rel, on the other hand, a state that is otherwise not reachable while increasing processor allocation can also be reached while releasing processors.

Processor requirement: Figure 8(a) plots the maximum cumulative processor requirement throughout the duration of the trace as a function of D for: (1) an adaptive system with PAL along with the four policy combinations, (2)

(a) Lazy-Alloc (Eager-Rel and Lazy-Rel)

(b) Eager-Alloc (Eager-Rel and Lazy-Rel)

Figure 7: FSA for PAL for the v6/TCP service. The reverse transitions are not shown in the figure. With Eager-Rel policy, when the queue becomes empty, the reverse transitions from each state jump to n_{min}. With Lazy-Rel policy, the reverse transitions from each state can jump back to any states between j and n_{min} before reaching n_{min}.

(a)

(b)

Figure 8: (a) Provisioning required by various PAL variants, a static system, an ideal adaptive system and a reactive system. (b) Packets missing delay bound in a reactive system even when there are sufficient number of processors in the system. PAL does not miss bounds for any packets.

Static — a non-adaptive system, (3) No-Overhead — a hypothetical adaptive system for which $t_{sw} = 0$, and (4) a simple reactive system that adds one processor every time the queue crosses the threshold as in Equation 2. The graph demonstrates that when $D < t_{pkt} + t_{sw}$ ($= 220\mu$s), PAL with any policy combination requires significantly greater level of provisioning as compared to No-Overhead. In this region, PAL is speculative; it allocates processors even before the queue length reaches Q^j_{lim}. The difference between PAL and No-Overhead becomes smaller for $D > 700\mu$s. At this point, a static system uses 1.5 times more processors than PAL.

To compare PAL with a quantum based system, consider $D = 4 \times t_{sw}$ ($= 800\ \mu$s) in Figure 8(a). At this value of D all variants of PAL use only 4-13% extra processors in the worst case (note that the graph plots the maximum processor requirement) when compared to No-Overhead. This amounts to an efficiency of 87-96% in the worst-case. To provide the same efficiency, a quantum based system has to use a quantum size of $Q = 8 \times t_{sw}$ to $25 \times t_{sw}$, thereby providing only larger delay bounds of size Q. Note that efficiency of a quantum based system is defined as $1 - \frac{t_{sw}}{D}$.

For $D \in [220, 700]$, different variants of PAL (with different allocation and release policies) behave differently. First, the figure shows that PAL with Lazy-Alloc uses significantly higher number of processors than PAL with Eager-Alloc. This is because, after $D > t_{pkt} + t_{sw}$ ($= 220\ \mu$s), the FSA for Lazy-Alloc policy has higher values of k^j_a than Eager-Alloc. Further, with Lazy-Alloc, once $D \geq t_{pkt} + 2 \times t_{sw}$ ($= 420\mu$s), setting $\tau = -200\mu$s yields $n_{min} = 0$, and a high value of k^0_a. Hence, Lazy-Alloc requires significantly high number of processors at $D = 420\mu$s. Note that, because of such frequent re-allocations between 0 and k^0_a, the processor requirement for Lazy-Alloc can exceed that of a static system. Second, for a given allocation policy, Eager-Rel policy uses greater number of processors than Lazy-Rel policy. This is because once the queue becomes empty, Eager-Rel policy releases all but n_{min} processors, whereas Lazy-Rel releases processors based on the current arrival rate of packets. As a result, Lazy-Rel allows several other reachable states in the FSA, thereby better matching the processor allocation to the workload requirements.

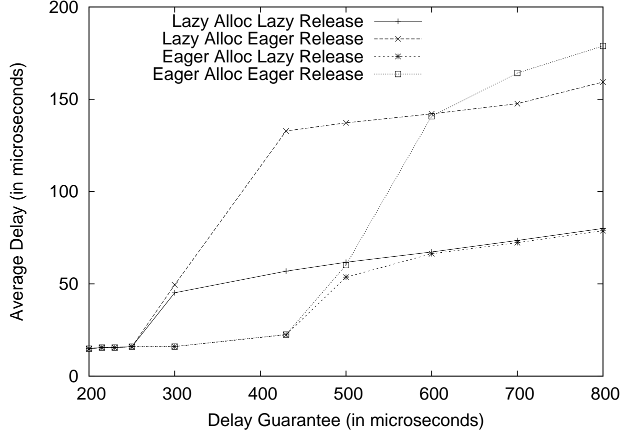

Figure 9: Delay properties of different variants of PAL.

With increase in D, k_a^j for all the policies decreases, thereby increasing the number of reachable states in the respective FSAs; hence the difference between the policies decreases.

Comparison to a reactive system: Figure 8(a) also shows the provisioning required by a reactive system. While the reactive system requires lower provisioning than PAL, Figure 8(b) shows that the reactive system causes packets to miss their delay bound. The number of packets that miss the bound increases with t_{sw}. This is because the derivation of the threshold (Equation 2) ignores t_{sw} and as a result assumes that j^{th} processor added previously is already processing packets when the queue length reaches Q_{lim}^j.

Delay properties: To study the delay properties of PAL with different allocation/release policies, we measure the average delay seen by the packets while varying D. Figure 9 plots the variation in this average delay as a function of D. It shows that the average packet delay observed with all policies is substantially lower than the delay bound D. The average delay is worse for Eager-Rel than Lazy-Rel because Eager-Rel aggressively releases processors, thereby paying the cost in terms of delay when allocating the processors again. Lazy-Alloc has larger delay than Eager-Alloc because the former delays the allocation of processors by τ during every allocation.

4.2.2 Robustness

The ability to match processor allocations to the fluctuating traffic requirements makes an adaptive system more robust than a statically provisioned system. Traditionally, static provisioning for each service is determined based on a trace of packets collected from the location that the system is deployed. However, the workload can deviate from the trace with which the system is provisioned in two ways (1) the cumulative processor requirement at any instant can exceed the provisioning in the system, (2) the workload mix for the services can be different at different times, although the cumulative requirement is lower than the provisioning. In such scenarios, the static system can miss deadlines of packets. Whereas, an adaptive system tries to match the processor allocations to fluctuating requirements, thereby reducing the number of packets that miss their deadlines.

In this section, we demonstrate the robustness of an adaptive system using PAL by measuring the number of packets that miss their delay bound when we vary the total processor provisioning in the system. Through trace analysis, we define the the maximum processor requirement of each service in order to meet delay bounds of packets. Given a certain amount of processor provisioning, the static system allocates the processors in the ratio of the maximum requirement of each service. We define the static system's allocation as the *core requirement* for each service. The adaptive system uses PAL to determine the processor requirement for each service, and coordinates the processor allocation among services by passing each service's allocation requests through the coordinator (Figure 2). The coordinator's behavior is summarized as follows.

- When the current cumulative processor requirement of all services is less than the provisioning, the coordinator allocates the processors in the ratio of the current requirement of each service.
- When the cumulative requirement exceeds the provisioning (i.e., the system is overloaded), the coordinator allocates processors such that the minimum of each service's core requirement and current requirement is first met. Then, to satisfy the extra requirement of services, the coordinator divides the rest of the processors (if any) in the ratio of the current requirement.

Note that when the current requirement of each service exceeds its core requirement, the adaptive system behaves like the static system.

Figure 10(a) plots the number of packets that miss the delay bound in an adaptive and a statically provisioned system as the total processor provisioning in the system is varied. We set $D = 600\mu s$. The adaptive system uses Lazy-Alloc-Lazy-Rel policy. We make two observations here. First, for a given provisioning, the number of packets that miss their delay bound in a static system can be orders of magnitude higher than an adaptive system. For example, (consider a vertical line at x=20) when the number of processors in the system is 20, 0.2% of the packets miss their delay bound in a static system, as opposed to 0.003% in an adaptive system. Second, in order to meet delay bound for the same number of packets, an adaptive system requires fewer processors than a static system. For example, (consider a horizontal line at y = 0.0001) a static system requires 50% more processors than an adaptive system to achieve a delay bound of $D = 600\mu s$ for 99.99% of the packets.

To illustrate how fragile static provisioning can be, we determine the static provisioning for each service with one trace UNC1, and feed a different trace UNC2 (collected at a different time from the same location) to our simulator. Figure 10(b) shows that the static system loses substantially more packets with mismatch in provisioning. This graph demonstrates that the workload can differ substantially from

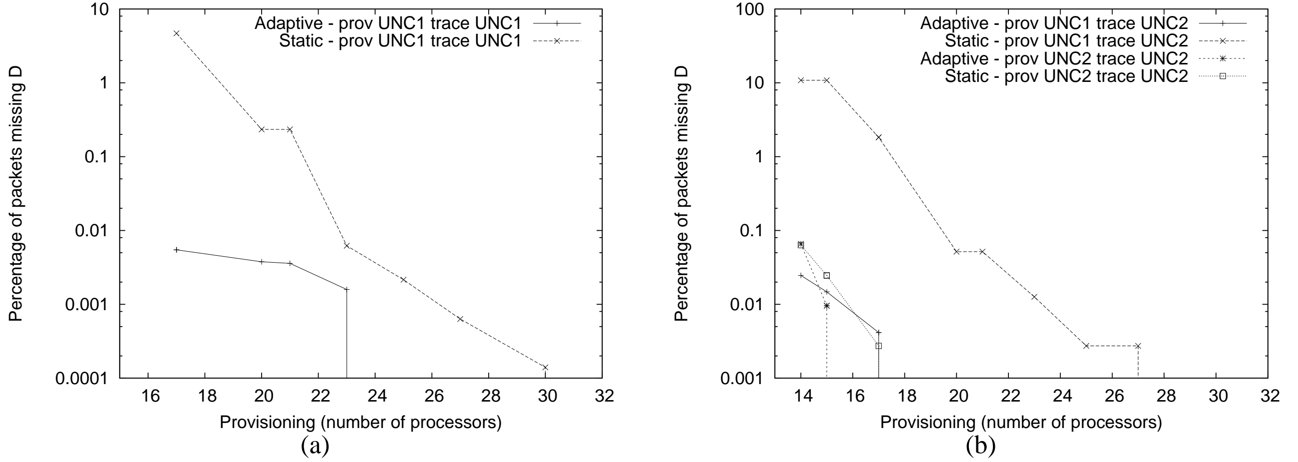

Figure 10: Benefits of adaptation. $D = 600\mu s$.

the trace used to provision a static system; in such scenarios, a static system behaves poorly. In contrast, an adaptive system causes very few packets to miss delay bound even with mismatches in provisioning.

The benefits of adapting processor allocations dynamically have implications on the design of robust packet processing systems. In practice, (1) it is difficult to obtain a trace that is representative of all possible traffic mixes of services, and (2) provisioning statically for the possible worst-case requirement of each service can be prohibitively expensive. Hence, the capability to adapt processor allocations simplifies the task of designing robust systems.

4.2.3 Sensitivity to t_{sw}

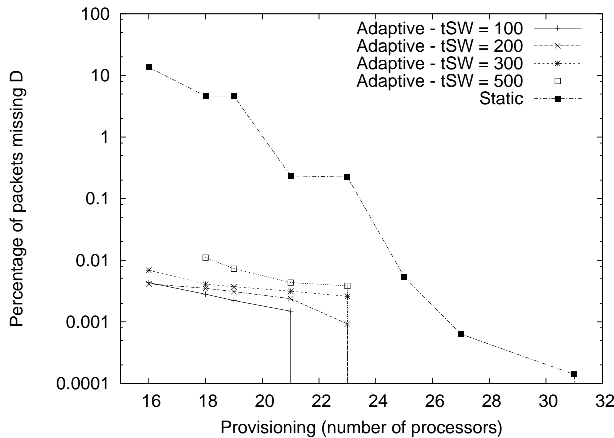

Figure 11: Sensitivity to t_{sw}. $D = 800$.

Figure 11 shows the sensitivity of adaptation benefits to the context switch overhead. The graph shows that the benefits of adaptation reduce with increasing t_{sw} (as expected), but the benefits can still be substantial compared to a static system.

5 Related Work

This paper presents a processor allocation algorithm that effectively multiplexes processors at fine time-scales in a packet processing system to satisfy application performance requirements, while coping well with the fluctuating workloads and significant context-switch overheads.

The problem of adapting the allocation of processors dynamically at fine time-scales has been explored in various domains. These solutions can be categorized into *request-level scheduling* [8, 29, 33] and *quantum-based scheduling* [7, 13, 15, 24, 31] algorithms. Request-level scheduling algorithms allocate processors to requests in the strict order of a scheduling parameter (e.g., the deadline assigned to the request). Such algorithms may switch a processor from providing one service to another on every request. In the case of packet processing systems, the processing time of a packet t_{pkt} is often significantly smaller than the processor context-switch overhead t_{sw}. Hence, such request-level scheduling algorithms can waste significant resources in transitioning processors from one service to another.

Quantum-based scheduling algorithms address the limitation of request-level schedulers by allowing a service to process a *fixed* number of requests prior to switching the processor allocation to a different service. These algorithms amortize the context-switch overhead across multiple requests; however, they can't provide delay bound smaller than the time quantum Q. For packet processing systems, delay bounds similar in magnitude as t_{sw} are desirable; hence, fixed quantum-based algorithms are not well-suited for this environment.

Several feedback-based systems built for servers achieve variable size quantum when adapting resource allocations. Larus et.al. [20] achieve the benefits of instruction cache locality in staged-servers, by keeping processors assigned to a stage as long as there is work for the stage to do. However, they do not consider latency-sensitive applications. Several solutions detect the requirement of processor allocation based on increased response times of re-

quests [2, 22, 32]. Instead of using such late feedback and missing delay bounds of packets, we monitor packet queue lengths to detect early the need for processor allocation.

Many solutions have been proposed to perform coarse-grain multiplexing of resources in shared hosting centers [4, 9, 10, 21, 28]. These solutions are designed to adapt resources for handling diurnal fluctuations of traffic, and are not suitable to provide delay bounds on packets in the packet processing domain. Further, many of these solutions design a policy to maximize the benefits of resource management, and then evaluate the impact of the policy on the system performance (e.g. response times). Instead, we take the reverse approach and derive an adaptation algorithm from the performance constraints.

6 Conclusion

We present a novel delay-conscious processor allocation algorithm (PAL) for packet processing systems. PAL adapts the allocation of processors among packet processing services hosted by a system to match the workload requirements and minimize the number of packets that miss a configured delay bound, while taking into account the processor reconfiguration overheads. We demonstrate through trace-driven simulations that an adaptive system using PAL is robust to traffic fluctuations for a given processor provisioning level—it can reduce the number of packets that miss the delay bound by orders of magnitude compared to a statically provisioned system. Further, a system with static allocation of processors to services requires 1.5-2 times the number of processors compared to an adaptive system.

References

[1] Intel IXP Family of Network Processors. http://www.intel.com/design/network/products/npfamily/index.htm.

[2] 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*, 13(1):80–96, 2002.

[3] M. Adiletta, D. Hooper, and M. Wilde. Packet Over SONET: Achieving 10 Gigabit/sec Packet Processing with IXP2800. *Intel Technology Journal*, 6(3), 2002.

[4] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano-sla based management of a computing utility. In *IEEE/IFIP Integrated Network Management Proceedings*, 2001.

[5] 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.

[6] R. Berrendorf and B. Mohr. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. http://www.fz-juelich.de/zam/PCL/doc/pcl/pcl.pdf.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI*, 2000.

[8] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *IEEE Real Time Technology and Applications Symposium*, 2001.

[9] 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.

[10] J. S. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[11] Cisco Systems Inc. Implementing NAT Protocol Translation. `http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123cgcr/ipv6_c/sa_natpt.htm`.

[12] D. Decasper, Z. Dittia, G. M. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of ACM SIGCOMM*, 1998.

[13] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[14] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1), 1991.

[15] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Joint international conference on Measurement and modeling of computer systems*, 2004.

[16] S. Karlin and L. Peterson. VERA: an extensible router architecture. *Computer Networks*, 38(3), 2002.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[18] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin. A Case for Run-time Adaptation in Packet Processing Systems. In *Proc. of the HOTNETS-II Workshop.*, November 2003.

[19] M. E. Kounavis, A. T. Campbell, S. T. Chou, and J. Vicente. Programming the Data Path in Network Processor-Based Routers. *Software Practice and Experience*, 2004.

[20] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *USENIX Annual Technical Conference*, 2002.

[21] P. W. K. Lie, J. C. S. Lui, and L. Golubchik. Threshold-Based Dynamic Replication in Large-Scale Video-on-Demand Systems. *Multimedia Tools and Applications*, 11(1), 2000.

[22] P. Pradhan, R. Tewari, S. Sahu, C. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. In *Int. Workshop on Quality of Service*, 2002.

[23] Y. Qiao, J. Skicewicz, and P. Dinda. Multiscale Predictability of Network Traffic. Northwestern University. Technical report.

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

[25] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *Proceedings of the 2nd Workshop on Network Processors (NP-2)*, February 2003.

[26] 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.

[27] 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.

[28] S.Ranjan, J.Rolia, H.Fu, and E.Knightly. QoS-Driven Server Migration for Internet Data Centers. In *Int. Workshop on Quality of Service*, 2003.

[29] 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.

[30] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, February 2000.

[31] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[33] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, 1999.

[34] Z. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-Time Scaling Behaviors of Internet Backbone Traffic: An Empirical Study. In *Proceedings of the IEEE INFOCOM.*, 2003.