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

The functionality of packet processing applications is often partitioned into pipeline stages; these stages are allocated a subset of the multiple processors available in a packet processing system. The workload, and hence the processing requirement, for each pipeline stage fluctuates over time. Adapting processor allocations to pipeline stages at run-time can improve robustness of the system to traffic fluctuations, can reduce processor provisioning requirement of the system, and can conserve energy. In this paper, we present an on-line algorithm for adapting processor allocations while ensuring that the additional delay suffered by packets as a result of adaptation is deterministically bounded. The resulting Processor Allocation Algorithm (PAL) is simple, but it allocates only as many processors to stages as needed to meet packet delay guarantees, accounts for system reconfiguration overheads, and copes 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 that for a set of trace workloads PAL can reduce processor provisioning level by 30-50%, reduce energy consumption by 60-70% while increasing the average packet processing delay by less than 150μs. We describe our prototype implementation for Intel's IXP2400-based packet processing system.

1 Introduction

Adapting processor allocations to pipeline stages of a packet processing application at run-time can improve robustness of the system to traffic fluctuations, can reduce processor provisioning requirement of the system, and can conserve energy. In this paper, we design, implement, and evaluate an adaptive processor allocation algorithm for packet processing systems. In what follows, first we discuss the background, the problem/opportunity, and the challenges in designing an adaptive processor allocation algorithm for packet processing systems. Then, we outline the contributions of this research.

Background Packet processing systems (PPS) are designed to process network packets efficiently. Over the past several years, the diversity and complexity of applications supported by PPS have increased dramatically. Examples of applications supported by PPS include Virtual Private Network (VPN), intrusion detection, content-based load balancing, and protocol gateways. Most of these applications are specified as graphs of functions and the specific sequence of functions invoked for a packet depends on the packet's type (determined based on the packet header and/or payload) [19, 21].

For most of these packet processing applications, the time to process a packet is dominated by memory-access latencies. Hence, an architecture containing a single, high-performance processor is often not suitable for a PPS. To mask memory access latencies, and thereby process packets at high rates, most modern PPSs utilize multiple parallel processors. For instance, Intel's IXP2800 network processor—a building block used in a wide-range of PPSs—includes 16 RISC cores (referred to as *micro-engines*) and an XScale controller.

To achieve high packet processing throughput in such multi-processor environments, it is essential that the code fragments used to process packets reside in instruction caches. This is because, each packet processing application can be thought of as a large *loop* that repeats for every packet; to ensure high packet processing throughput, the entire loop body must fit into the instruction cache. Today's processors or cores within network processors, however, are configured with a very limited size instruction store/caches (e.g., 4K instructions in Intel®'s IXP2800 network processor); the limited instruction store is often sufficient to hold code for a portion of the application, but rarely enough to hold code for the entire application. This leads to software designs in which the responsibility for processing packets is partitioned into a set of pipeline stages; further, the stages are mapped onto processors—with each processor specialized to perform one task [1]. Partitioning applications into pipelined stages is also important for improving robustness of request-processing systems [38].

The Problem/Opportunity Today, the allocation of processors to pipeline stages of an application is done statically (at design-time). Consequently, to guarantee robustness to fluctuations in the arrival rate for different types of packets, packet processing systems often provision sufficient number of processors to handle the expected *maximum* load for each pipeline stage. However, as illustrated by Figure 1, the observed load fluctuates significantly over time and at any instant is often substantially lower than the maximum load [22, 25, 27, 29, 40].

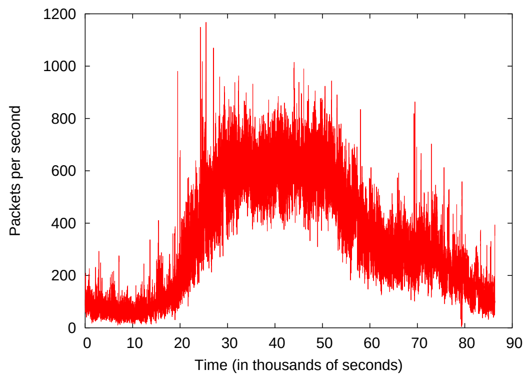


Figure 1: Packet arrivals per second over a day for the Auckland trace [26]

In such settings, an adaptive run-time environment—that can change the allocation of processors to pipeline stages at run-time—can yield significant benefits. First, the ability to match processor allocations to the processing demands for each pipeline stage leads to system designs that are *robust* to traffic fluctuations. An adaptive system can allocate appropriate number of processors to each stage even when the processing demands for a stage exceed design-time expectations, as long as the cumulative demands do not exceed the provisioning level; further, this simplifies the determination of the processor provisioning level for the entire system. Second, by multiplexing processors among different types of packets, an adaptive system can reduce the cumulative processor requirement (or provisioning level), and thereby reduce system cost. Finally, by reducing the power consumption of idle processors (e.g., by turning off processors or running them in low-power mode), an adaptive system can conserve energy.

The Challenges Although the properties of network workloads and of packet processing hardware raise opportunities for adaptive processor allocations, they also raise key challenges. First, because network traffic can fluctuate at multiple time-scales, accurately predicting traffic arrival patterns is difficult [25, 27, 29, 40] and intervals of idleness or low load may often be short [22], so it may be difficult to take advantage of periods of low load. Second, allocating and releasing processors incurs delay/overhead (generally of the order of a few hundred microseconds [23]). If the system releases processors too aggressively during idle periods, then because of the inherent delay in re-allocating processors, a burst of arriving packets might suffer unacceptable delays or losses.

Our Contributions In this paper, we present an on-line algorithm for adapting processor allocations while ensuring that the additional delay suffered by packets as a result of adaptation is *deterministically bounded*. Our Processor Allocation Algorithm (PAL) is simple, but it allocates only as many processors to stages as needed to meet packet delay guarantees, accounts for system reconfiguration overheads, and copes with the unpredictability of packet arrival patterns. PAL, like active queue management algorithms [6, 9, 15], makes processor allocation/release decisions based only on the current queue length; it does not rely on any predictions for future arrival patterns beyond knowing the worst-case arrival rate.

- Given a current allocation of j processors for a stage and a worst-case delay bound D, PAL allocates additional processors only when the current allocation is unable to process within D the sum of (a) all currently enqueued incoming packets and (b) the maximum number of packets that could arrive during a processor's allocation latency. Surprisingly, for realistic system configurations, a simple sufficient condition to meet this general activation requirement is to activate the $j+1$st processor when the set of enqueued packets first exceeds the number of packets that j processors can process within D.

- Conversely, when the system has excess capacity, PAL releases one or more processors when both (a) the input queue becomes empty and (b) the minimum time until the processors would be reactivated under a worst-case arrival rate exceeds the latency for allocating/releasing a processor.

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.

There are four salient features of PAL. First, PAL offers the flexibility to instantiate various policies for *eager* or *lazy* allocation and release of processors; this allows PAL to tradeoff adaptation frequency/benefits with the delay incurred by packets. Second, we show that for acceptable values of the delay bound D, total processor requirement for PAL is within 10-30% of an *ideal, hypothetical* setting that incurs no overhead for processor allocation/release. Third, PAL does not require prediction of future packet arrival patterns. Given the variability of packet arrival rates in many network environments [25, 27, 29, 40], algorithms that do not depend on on-line prediction are likely to be less complex and more effective than those that do. Fourth, PAL deterministically meets a configurable bound on packet processing delay.

We have evaluated PAL through simulations; further, we have implemented a prototype adaptive processor allocation framework for a packet processing system based on Intel's IXP2400 network processor. Using simulations, we demonstrate that for a set of trace workloads PAL can reduce processor provisioning level by 30-50%, reduce energy consumption by 60-70% while increasing the average packet processing delay by less than 150μs.

The rest of the paper is organized as follows. In Section 2, we describe our system model. In Sections 3, we discuss our processor allocation algorithm. We describe the results of our experimental evaluation in Section 4, and discuss our prototype implementation in Section 5. Related work is discussed in Section 6, and finally, Section 7 summarizes our contributions.

2 System Model

We consider a packet processing system (PPS) with $\mathcal{P}$ processors, and an application with $\mathcal{S}$ pipeline stages. At any instant, a subset of the processors are allocated to each pipeline stage. A packet arriving into the system is processed by a subset of pipeline stages prior to departure. Each pipeline stage is associated with a queue; packets are queued into the stage's queue until a processor becomes available to service the packet (see Figure 2). We assume that a packet queued for service at stage i can be served by any of the processors allocated to stage i. Let the time taken to service a packet at stage i be t^i_{pkt} units; and each stage service packets in the order of their arrival. On being processed by a stage, the packet is queued either for processing at the next processing stage or for transmission at an outgoing link (once the packet has been processed by all the required stages).

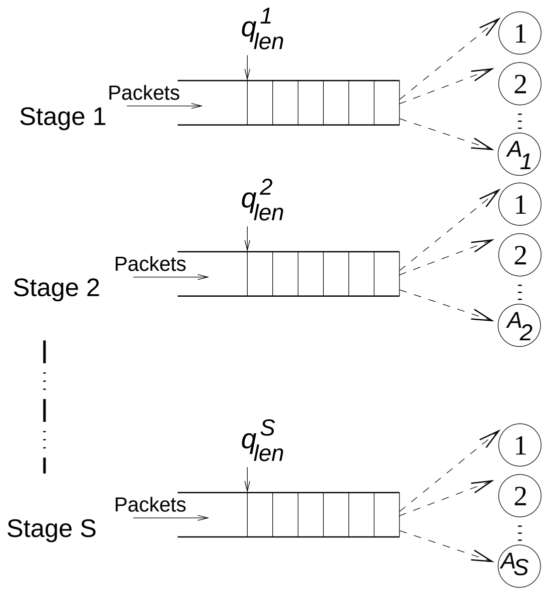


Figure 2: System model. $\mathcal{A}_i, i \in [1, \mathcal{S}]$ denote the number of processors allocated to pipeline stage i

Let the delay between when a packet is enqueued and when its processing is complete by stage i be bounded by D^i. The arrival rate of packets into each queue may fluctuate over time; hence, the allocation of processors to pipeline stages changes over time. Let the maximum rate of packet arrivals into queue for stage i be given by R^i_{arr}, and let N^i_p denote the maximum number of processors required to process the worst-case arrival rate R^i_{arr} within the delay bound D^i. Observe that for a non-adaptive system, $\mathcal{P} = \sum_{i=1}^{\mathcal{S}} N^i_p$. An adaptive system can multiplex processors among different pipeline stages; this facilitates a system with provisioning level $\mathcal{P} < \sum_{i=1}^{\mathcal{S}} N^i_p$ to provide deterministic delay bounds D^i to each packet processed by stage i. We assume that allocating and releasing a processor incurs a delay t_{sw}. Table 1 summarizes these system model parameters.

Parameter	Description
$\mathcal{P}$	Number of processors in the system
$\mathcal{S}$	Number of pipeline stages
R^i_{arr}	Worst-case arrival rate for stage i
N^i_p	Worst-case processor requirement for stage i
t^i_{pkt}	Processing time per packet for stage i
t_{sw}	Switching delay (allocate/release processor)
D^i	Delay guarantee for stage i

Table 1: System model parameters

Throughout the paper, we consider a system provisioning level such that the total instantaneous processor demands for all pipeline stages never exceeds the provisioned capacity. This requirement is essential to provide deterministic bounds on the delay experienced by packets at each pipeline stage. Further, 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.

3 Processor Allocation Algorithm

For each pipeline stage, at any instant, an adaptive system should determine and allocate only as many processors as needed to process packets before their processing deadlines. Adapting processor allocations dynamically has three benefits. First, the ability to match processor allocations to the processing demands for each pipeline stage leads to system designs that are *robust* to traffic fluctuations. An adaptive system can allocate appropriate number of processors to each stage even when the processing demands for a stage exceeds design-time expectations, as long as the cumulative demands do not exceed the provisioning level; further, this simplifies the determination of the processor provisioning level for the system. Second, adaptation enables statistical multiplexing of processors among pipeline stages, which, in turn, reduces the overall processor provisioning requirement. Finally, by deactivat-

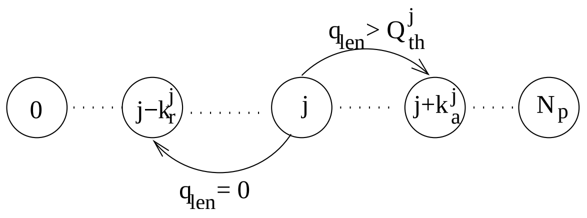


Figure 3: The finite-state automaton for PAL. The quantities in each state denote the current processor allocation.

ing (or running in low-power mode) spare processors, the adaptive system can reduce overall energy consumption.

Our Processor Allocation Algorithm (PAL) maintains for each pipeline stage a *finite state automaton (FSA)* (see Figure 3). Each state in the FSA for pipeline stage i represents a processor allocation level for stage i; state transitions denote processor allocation and release events. State transitions are triggered based on the length q^i_{len} of the queue for stage i.

When stage 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 stage 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 stage 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 pipeline stages, we present the FSA construction for a single pipeline stage. Further, for brevity, we will eliminate any reference to a specific stage (and drop superscript i from all symbols defined in Table 1) from our discussion.

3.1 Processor Allocation

3.1.1 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 rate at which packets are serviced from the queue is a function of the number of processors allocated to the stage. 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:

$$R^j_{dep} = \frac{j}{t_{pkt}} \quad (1)$$

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 pipeline stage is allocated j processors, then the total number of packets in the stage is $Q^j_{lim} + j$. Hence, Q^j_{lim} is determined by

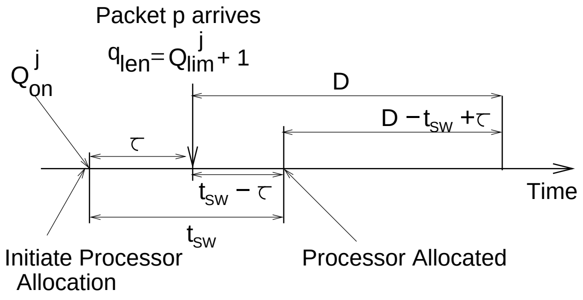


Figure 4: Allocation procedure: Timing diagram

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

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 stage 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.

Let us assume that the system requests allocation of an additional processor τ units of time *prior* to the arrival of packet p (see Figure 4). $\tau > 0$ represents a *speculative* system that allocates additional processors in anticipation of q_{len} exceeding Q^j_{lim}; in contrast, $\tau \leq 0$ represents a *reactive* system that allocates additional processors only after $q_{len} > Q^j_{lim}$.

Observe that a newly allocated processor can serve packets only after t_{sw} time units. Hence, 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

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

This constraint requires

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

which lets 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 after the queue length $q_{len} > Q^j_{lim}$, 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 $q_{len} > Q^j_{lim}$ and thereby trigger the allocation of additional processors when $q_{len} > Q^j_{lim} - \tau \times (R_{arr} - R^j_{dep})$. Hence, if $(t_{pkt} + t_{sw} - D) > 0$, the $Q^j_{th} = Q^j_{lim} - \tau \times (R_{arr} - R^j_{dep})$.*

This is an important conclusion; it indicates that when the delay bound $D \geq t_{pkt} + t_{sw}$, PAL can be completely reactive; it can observe the queue build up and react only upon receiving a packet whose delay guarantee will be violated. We expect that the condition $D \geq t_{pkt} + t_{sw}$ is likely to be met by most realistic system configurations. This is because, even for a non-adaptive system, the delay bound D must be at least t_{pkt} (the time required to process a packet); increasing the delay bound further enables a system to achieve the benefits of adaptive resource allocation.

3.1.2 k^j_a: How Many Processors to Allocate?

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 stage queue at a rate no greater than 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:

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

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

This leads to the following conclusion.

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

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

where N_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 ($j \in [0, N_p]$), 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 $j = N_p$, no additional processors can be allocated; hence, $k^{N_p}_a = 0$.

- 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:

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

  Observe that packet processing systems are often provisioned to meet the demands of the expected maximum arrival rate R_{arr}. In such an appropriately provisioned system, the number of processors N_p is, in fact, equal to $R_{arr} \times t_{pkt}$. In such a case, for all values of j, $k^j_a = N_p - j$. Thus, the FSA contains a direct transition from every state to the state with N_p allocated processors; every trigger to increase processor allocation will request all available processors.

  Selecting $\tau > t_{pkt} + t_{sw} - D$ results in smaller values of k^j_a. The greater the value of τ, the smaller the value of k^j_a. Thus, k^j_a 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^j_a as compared the case when $\tau = 0$. Smaller values of k^j_a are preferable; they allow PAL to gradually increase processor allocations and thereby better utilize available processors across different pipeline stages.

- If $D + \tau > (R_{arr} \times t_{pkt} \times t_{sw} + t_{pkt})$, then from Equation (5), for all values of $j < N_p$, $k^j_a = 1$; thus, processors will be allocated one at a time. With this, the number of reachable states in the FSA becomes equal

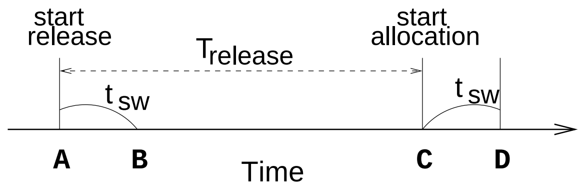


Figure 5: Condition for release to be beneficial

to N_p. The greater are the number of reachable states in the FSA, the better is the opportunity for PAL to align processor allocation to processing demand.

3.2 Processor Release

3.2.1 When to Release Processors?

Processors allocated to a stage should be released when the processors are running at low-levels of utilization. This would be the case when the packet processing capacity allocated to a stage (and hence the maximum packet service rate) exceeds the packet arrival rate for the stage significantly. Instead of monitoring the processor utilization levels continuously, PAL simply estimates the possibility of over-allocation by monitoring queue length. In particular, PAL uses the condition $q_{len} = 0$ as a trigger to release an appropriate number of processors.

3.2.2 k_r^j: How Many Processors to Release?

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 guarantee 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 stage 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 maximum 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:

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

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

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

This 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 guarantee is met for all packets. Hence, the number of processor that can be released from state j is bounded by:*

$$k_r^j \leq j - n_{min} \tag{9}$$

We make the following observations.

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

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

  Thus, 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.*

3.3 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.

3.3.1 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 be completely reactive; it 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 pipeline stage 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), $\forall j : k_a^j = N_p - j$. Since $n_{min} = 0$, this results in a two-state FSA – the two states represent allocations of 0 and N_p processors to the pipeline stage. 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 supports a multi-state FSA. For $D > t_{pkt} + 2 \times t_{sw}$, this can be achieved by selecting $\tau = -t_{sw}$.

3.3.2 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 stages. However, it also introduces a significant number of allocation/release transitions in the system (thereby adversely affecting the stability 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 stage. 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 $\frac{\widehat{R}_{arr}(\Delta)}{1/t_{pkt}} = \widehat{R}_{arr}(\Delta) \times t_{pkt}$ denotes the number of processors required to match the packet arrival rate at a stage. 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 (and thereby leads to a stable system design).

4 Experimental Evaluation

In this section, we first describe our experimental methodology, and then present results of our simulations.

4.1 Experimental Methodology

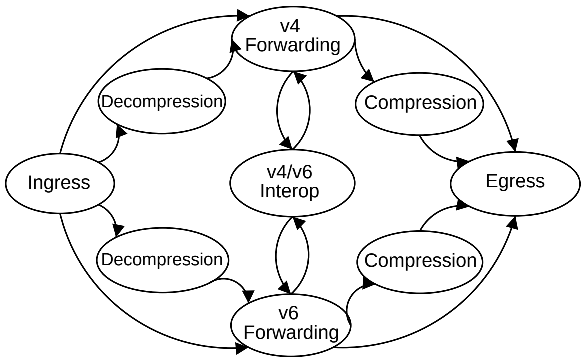


Figure 6: 3G wireless router architecture

Application Model: We conduct all our experiments using the model of a 3G wireless router— a canonical packet processing application. Figure 6 shows the application graph with different packet processing stages. In this application, each incoming packet is sent to a link layer demultiplexor that identifies IPv4/IPv6 packets and determines whether or not the packet header is compressed. Packets with compressed headers undergo an appropriate decompression. The packets then go through appropriate IP forwarding that determines the next hop for the packet. The next hop address determines whether the packet should be simply forwarded, compressed and forwarded, or converted to the other version of the protocol. Table 2 summarizes the t_{pkt}^i values for each of these packet processing stages; we do not model Ingress and Egress stages for our analysis (since these stages often require dedicated processors.

Traces: We analyze traces collected from various points in the Internet. For brevity, we only discuss two traces – a 6 hour subset of NLANR trace (AUCKLAND) containing 20 million packets collected from a link connect-

Stage	t^i_{pkt} (in μs)
Interop IPv4 to IPv6	135
Interop IPv6 to IPv4	32
IPv4 forward	55
IPv6 forward	55
IPv4/v6 compression	59
IPv4/v6 decompression	24

Table 2: A simple 3G wireless work model.

ing New Zealand to US [26], and an 8 minute long trace with 30 million packets collected from the high-speed link connecting University of North Carolina at Chapel Hill (UNC) to its network service provider [31].

We use these traces to estimate fluctuations in the arrival for each application stage as follows. Each of these raw traces consists of two packet sequences—incoming and outgoing. These two sequences represent packet arrivals of two different types. For the 3G-wireless router, all incoming packets are considered IPv4 packets, while all outgoing packets are considered IPv6 packets. Further, we assume that, with equal probability, all packets of each arriving flow have a compressed or an uncompressed header, all packets of each departing flow have a compressed or an uncompressed header, and all packets of each departing flow leave as is or are translated to the other version of the protocol.

Provisioning: We provision the system with $\sum_{i=1}^{\mathcal{S}} N^i_p$ processors, where N^i_p ($i \in [1,\mathcal{S}]$) denotes the maximum processor requirement for stage i. This provisioning level guarantees that a non-adaptive system can also meet the delay bounds for each packet. We determine N^i_p as follows. First, we derive, through trace analysis, $R^i_{arr}(D^i)$, the maximum rate of packet arrivals over intervals of length D^i. Since a single processor can process packets at the rate of $1/t^i_{pkt}$, N^i_p is derived as:

$$N^i_p = \left\lceil \frac{R^i_{arr}(D^i)}{1/t^i_{pkt}} \right\rceil \tag{12}$$

We make two observations. First, this approach for deriving processor provisioning level is conservative (and hence, favors the non-adaptive system); it derives the provisioning requirement for a particular trace. In practice, processor provisioning is determined based on the requirement to handle worst-case packet arrivals (perhaps at the line rate), which could be substantially higher than R^i_{arr} observed within a trace. Second, this definition of provisioning requires a smaller number of processors for the same workload with increasing delay bound D.

For our simulations, we assume that all stages have the same delay bound. We use t_{sw} = 200μs to allocate/release a processor [23].

```
D: 500   Np: 7  nmin: 2
[0] 7  [1] 6  [2] 5  [3] 4  [4] 3  [5] 2  [6] 1  [7] 0
-------------------------------------------------------
D: 1000  Np: 6  nmin: 0
[0] 3  [1] 3  [2] 2  [3] 2  [4] 2  [5] 1  [6] 0
-------------------------------------------------------
D: 2000  Np: 4  nmin: 0
[0] 2  [1] 1  [2] 1  [3] 1  [4] 0
```

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

```
D: 500   Np: 7  nmin: 4
[0] 4  [1] 4  [2] 3  [3] 3  [4] 2  [5] 2  [6] 1  [7] 0
-------------------------------------------------------
D: 1000  Np: 6  nmin: 3
[0] 2  [1] 2  [2] 2  [3] 2  [4] 1  [5] 1  [6] 0
-------------------------------------------------------
D: 2000  Np: 4  nmin: 2
[0] 1  [1] 1  [2] 1  [3] 1  [4] 0
```

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

Figure 7: FSA for PAL for the IPv4 forwarding stage.

4.2 Multiplexing Benefits

To evaluate the benefits of adapting processor allocations on the required level of provisioning, we consider PAL with the following processor 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 $k^j_r = (j - n_{min})$; and (2) **Lazy-Rel** with $k^j_r = \min\left(j - n_{min},\ \ j - \left\lceil \frac{\widehat{R}_{arr}(\Delta)}{1/t_{pkt}} \right\rceil\right)$ (see Equation (11).

Figure 7 shows the automata for the IPv4 forwarding stage for different values of D and different combinations of PAL policies. The first line in each row shows the value of D, the corresponding processor provisioning level, and n_{min}. The second line in each row represents the FSA in the format – [j] k^j_a, where j is the number of currently allocated processors, and k^j_a is the number of processors that are allocated when $q_{len} > Q^j_{th}$. We make multiple observations from Figure 7. First, as D increases, the required processor provisioning (N_p) decreases. Second, as D increases, both n_{min} and k^j_a decrease. Third, n_{min} decreases slower for Eager-Alloc, while it quickly reaches zero with Lazy-Alloc. Fourth, since $\tau_{Lazy-Alloc} \leq \tau_{Eager-Alloc}$ for all values of D, k^j_a (for all j) for Eager-Alloc is never higher than Lazy-Alloc. Fifth, k^j_a reduces with increase in j. Observe that the FSAs are independent of whether we use Eager-Rel or Lazy-Rel policies; these policies only control the number of reachable states in the FSA. When the algorithm uses Eager-Rel, since $k^j_r = (j - n_{min})$, 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.

Benefits: Figure 8 plots the maximum number of processors allocated at any instant throughout the duration of the trace as a function of D for: (1) a non-adaptive system (with $\mathcal{P} = \sum_{i=1}^{S} N_p^i$); (2) an ideal (albeit hypothetical) adaptive system for which $t_{sw} = 0$; and (3) a realistic system (with $t_{sw} = 200\mu$s) with PAL along with the four policy combinations. It shows that PAL utilizes 30-50% fewer processors than the non-adaptive system; this indicates that an adaptive system using PAL can reduce the processor provisioning level by 30-50%. Figure 8 also demonstrates that at small values of D ($D < 500\mu$s), PAL requires significantly greater level of provisioning as compared to the ideal case (with $t_{sw} = 0$); however, the difference between the two cases becomes smaller than 10% for moderate to high values of D. This indicates that at moderate to high values of D ($D \geq 600\mu$s), the provisioning required by PAL approaches that of an ideal adaptive system.

Figure 8 also illustrates that different variants of PAL (with different allocation and release policies) behave differently at various values of D. Figure 8(a) shows that PAL with Eager-Alloc uses significantly fewer processors than PAL with Lazy-Alloc for most values of D. This is because of two reasons. First, the FSA for the Lazy-Alloc policy has higher values of k_a^j. Second, with Lazy-Alloc, once $D \geq 600\mu$s, setting $\tau = -200\mu$s yields $n_{min} = 0$. At $D = 600\mu$s, the FSA for Lazy-Alloc-Eager-Rel only contains two reachable states (0 and N_p); Lazy-Alloc-Lazy-Rel, on the other hand, adds several other reachable states. Hence, Lazy-Alloc-Lazy-Rel performs better than Lazy-Alloc-Eager-Rel. With increase in D, the FSAs for all of these policies become multi-state; hence, the difference between the policies reduces with D.

Figure 8(b) shows similar results for the AUCKLAND trace. It also demonstrates that PAL with Eager-Alloc performs well for all values of D (the lines for Eager-Alloc with both Eager-Rel and Lazy-Rel policies overlap). It also shows the benefits of using Lazy-Rel policy over Eager-Rel (see the lines for Lazy-Alloc-Lazy-Rel and Lazy-Alloc-Eager-Rel). It is noteworthy that because the AUCKLAND trace is sparse (and hence contains large intervals with very low packet arrival rates), the Lazy-Alloc policy (that yields smaller values of n_{min}) performs as well or better than the Eager-Alloc policy. For brevity, in the rest of this section, we will only consider results for the UNC trace.

Delay Properties: To study the delay properties of PAL with different allocation/release policies, we measured the average delay seen by packets at the IPv4-v6 interop stage. 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 (MAX-DELAY). The average delay is worse for Lazy-Alloc-Eager-Rel than all other variants, since this variant aggressively utilizes the delay guarantee by eagerly releasing and lazily allocating processors. Further, at $D = 600\mu$s, the average delay increases significantly. This is because, at this value of D, $n_{min} = 0$; hence, PAL with Eager-Rel policy releases all idle processors. PAL with the Eager-Alloc policy incurs very small delays because (1) n_{min} is non-zero, and (2) processors are allocated eagerly.

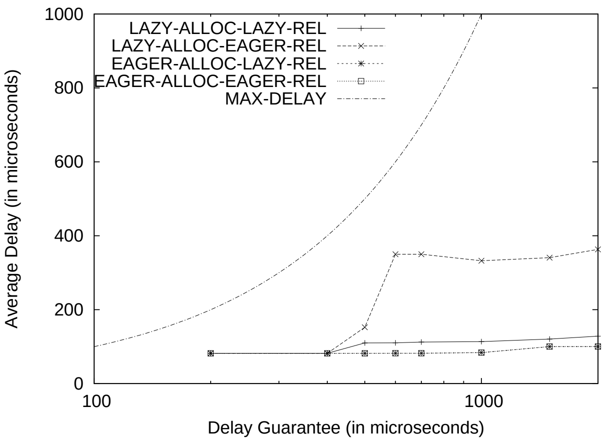
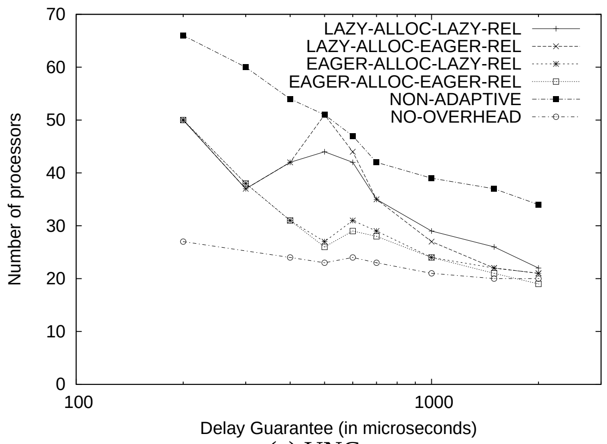


Figure 9: Comparison of delay seen by packets at the IPv4-v6 interop stage when using different algorithm variants for the UNC trace.

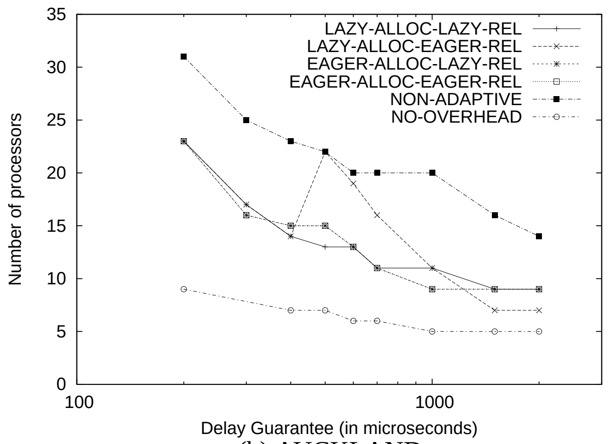
4.3 Energy Benefits

In this section, we derive the energy benefits resulting from deactivating (or running in a low-power mode) spare processors not allocated for any stage. We compare the energy benefits realized by PAL with those obtained by the *ideal* system (with $t_{sw} = 0$).

To derive the energy benefits, observe that transitioning a processor from deactivated (or low-power) state to an activated state is equivalent to allocating a processor to a stage; thus, the FSA for controlling processor activations is the same as the one for processor allocations derived earlier. The task of deactivating processors, however, is somewhat different from releasing an idle processor because a processor should be deactivated only if doing so saves energy. To formalize this condition, consider a processor that consumes P_{active} power when processing packets, and P_{idle} power while being idle. Let P_{sw} and t_{sw}, respectively, denote the power consumed while switching the processor state from activated to deactivated (and vice versa) as well as the time delay for such transitions. For simplicity, let us assume that the processor consumes no power in the deactivated state. In this case, the total amount of energy expended in transitioning $(j - n_{min})$

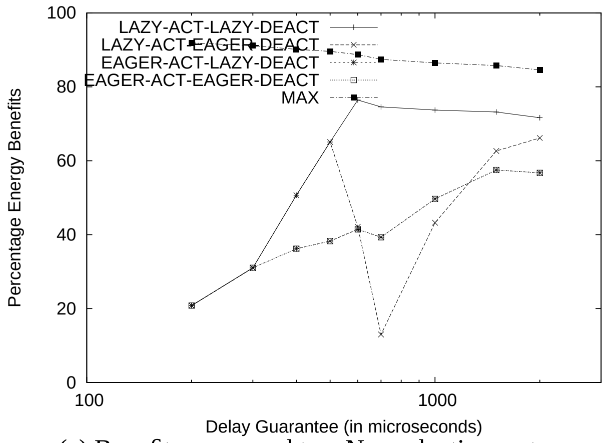


(a) UNC

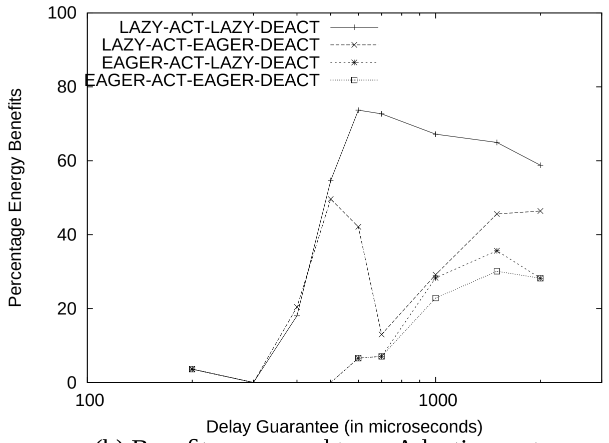


(b) AUCKLAND

Figure 8: Multiplexing benefits of different algorithm variants.



(a) Benefits compared to a Non-adaptive system



(b) Benefits compared to an Adaptive system with no energy management

Figure 10: Energy benefits of different algorithm variants.

processors from activated state to deactivated state and back is given by: $2 \times P_{sw} \times t_{sw} \times (j - n_{min})$. In comparison, by not deactivating $(j - n_{min})$ processors, the system would have expended $(T_{deact} + t_{sw}) \times P_{idle} \times (j - n_{min})$ energy, where T_{deact} is the time duration for which a processor remains deactivated. Thus, deactivating $(j - n_{min})$ processors conserves energy only if:

$$\begin{aligned} 2 \times P_{sw} \times t_{sw} \times (j - n_{min}) \quad &\leq \quad (T_{deact} + t_{sw}) \times P_{idle} \\ &\quad\quad \times (j - n_{min}) \end{aligned}$$

$$\Rightarrow T_{deact} \geq \left(2 \times \frac{P_{sw}}{P_{idle}} - 1\right) \times t_{sw} \qquad (13)$$

Using Equations (7) and (13), we derive a new bound for n_{min} as:

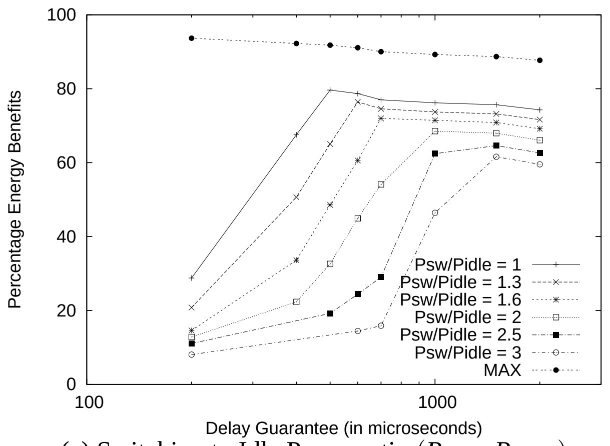
$$\forall j : n_{min} \quad \geq \quad \frac{(C \times R_{arr} - 1) \times t_{pkt}}{D - t_{pkt} + C} \qquad (14)$$

$$where\ \ C = \left(2 \times \frac{P_{sw}}{P_{idle}} - 1\right) \times t_{sw} + \tau$$

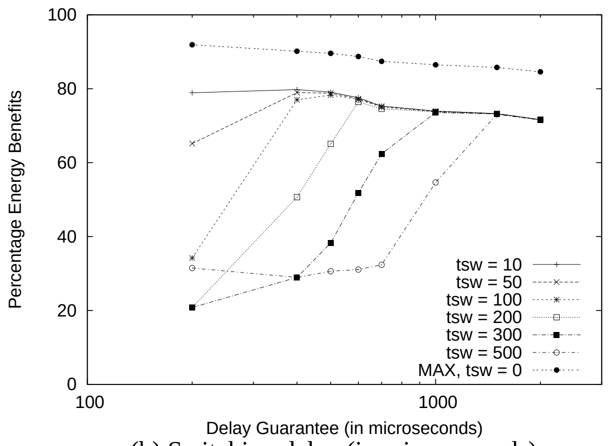
Using this value of n_{min}, we now consider PAL with its eager and lazy activation and deactivation policies.

System Parameters: For our experiments, we use the value for $\frac{P_{active}}{P_{idle}}$ and $\frac{P_{sw}}{P_{idle}}$ as 1.3 (the ratio of active to idle power for various IXP processors is between 1.2 and 1.3 [2]), and take $t_{sw} = 200\mu$s. We also study the effects of varying these parameters on the energy benefits of adaptation.

Metric: We compare the energy consumption of a system with adaptive processor activations with a non-adaptive system. In particular, for an adaptive algorithm A, we define the energy benefits over the non-adaptive system B as $1 - \frac{E_A}{E_B}$, where E_A and E_B, respectively, denote the energy consumed by the adaptive and the non-adaptive systems.



(a) Switching to Idle Power ratio $(P_{sw} = P_{active})$



(b) Switching delay (in microseconds)

Figure 11: Variation of power benefits with processor activation/deactivation delay and switching power for UNC.

Benefits: Figure 10 shows the energy benefits of PAL with different activation/deactivation policies over a non-adaptive system for different values of D. At small values of D, PAL yields low benefits for all the variants. This is because, at small values of D, n_{min} is large; hence, PAL maintains most of the processors in the activated state even though they are idle. However, with increase in D, n_{min} and k_a^j become small, and hence the benefits increase.

Notice that in Figure 10(a), the benefits using Lazy-Act-Eager-Deact drop significantly at $D = 700\mu$s and increase again. This is because, at $D = 700\mu$s, $n_{min} = 0$ and the FSA contains only two states. Hence, every time the queue becomes empty, PAL deactivates all the processors, and with the arrival of a new packet PAL re-activates all processors. Since UNC is a dense trace, the system thrashes by deactivating and activating processor back-and-forth without saving much energy. In the case of Eager-Act-Eager-Deact and Eager-Act-Lazy-Deact, n_{min} never reaches zero since $\tau = 0$. While these policies avoid the thrashing behavior of Lazy-Act-Eager-Deact, they are conservative in saving energy. The Lazy-Act-Lazy-Deact avoids the thrashing behavior by lazily deactivating processors, and yet achieves significant energy savings.

Figure 10(b) shows the energy benefits achieved because of processor activation/deactivation in a system provisioned with only the processors required by PAL. The basic behavior is the same as Figure 10(a); the magnitude of the benefits, however, is lower (as can be expected).

Sensitivity to system parameters: Figure 11 shows the effect of varying the processor switching overheads on energy benefits derived using PAL with the Lazy-Act-Lazy-Deact policy. Figure 11(a) shows that as $\frac{P_{sw}}{P_{idle}}$ and $\frac{P_{active}}{P_{idle}}$ increase, the benefits reduce. Given that most of today's processors have the ratio between 1 to 3, the graph shows that PAL can achieve significant benefits in systems with a variety of processors.

Figure 11(b) shows the effect of varying the processor activation and deactivation latency. It shows that as t_{sw} decreases, PAL can achieve significant benefits even for small values of D. Given the increasing focus on energy-aware system designs, we expect that the switching overheads will reduce over time. This result also indicates that the energy benefits can be increased by using shallower *sleep* states for processors. The shallower states incur lower wakeup latency but consume a small amount of energy. Our algorithm can be easily generalized to incorporate multiple sleep states; this generalization, however, is beyond the scope of this paper.

5 Prototype

We have implemented PAL in the context of the Shangri-La run-time system(RTS) [35][1]. The goal of Shangri-La is to create a programming environment for simplifying the development and deployment of packet processing applications onto complex, multi-processor architectures. The run-time system is a critical component of Shangri-La. In what follows, we briefly describe our hardware platform, and then outline the implementation of PAL in the context of Shangri-La RTS.

5.1 Hardware Platform

Our setup contains a PC hosting the Radisys ENP-2611 board with the Intel® 600MHz IXP2400 network processor. IXP2400 contains one XScale™ core and 8 micro-engines. Each micro-engine has an instruction store to hold 4K-40 bit instructions that are optimized for fast-path packet processing. The XScale core runs an embedded version of Linux [3], and is typically programmed to perform the control path functions such as stopping, starting

[1]http://www.cs.utexas.edu/users/vin/research/shangrila.shtml

and loading code into the instruction store of the micro-engines. The micro-engines, on the other hand, execute fast-path packet processing functions.

5.2 The Shangri-La Run-Time System

The Shangri-La RTS provides mechanisms to load and execute pipeline stages onto micro-engines, as well as appropriate sensors for monitoring traffic fluctuations, system performance, and energy consumption levels. The adaptation engine within the RTS utilizes these mechanisms and sensors to adapt processor allocations dynamically.

To simplify the design of the adaptation engine, the Shangri-La run-time system supports a *resource abstraction layer (RAL)* [35]. RAL provides an abstract interface to the underlying hardware resources such as micro-engines, memory modules, locks, hash/crypto units, etc. RAL makes the design of adaptation engine, RTS, and other higher level applications simple and portable. To support adaptation capability, RAL exports the following interfaces:

- `proc_unit::stop()` – for stopping a micro-engine;
- `proc_unit::load()` – for loading code into the micro-engine instruction store;
- `proc_unit::checkpoint()` – for bringing a stage and its data structures to a consistent state before stopping a micro-engine;
- `pkt_queue::get_length()` – for getting the length of the packet queue of a stage; and
- `pkt_queue::get_num_pkts()` – for determining the number of packets received in a queue since its creation.

The processing units and packet queues are identified using unique *handles*; during initialization, the RTS creates these handles and passes them to the adaptation engine. Information about the application—the stages, the corresponding code, appropriate communication channels between stages—as well as system parameters such as t^i_{pkt} and t_{sw} are provided by the user to the RTS at startup. The adaptation engine loads each stage onto n_{min} processors, and subsequently uses PAL to allocate and release processors dynamically.

6 Related Work

Recently, pipelining/staging has been used as a programming paradigm for complex applications. [17, 18, 20, 21, 38]. Using this paradigm, applications are decomposed into stages and stages are connected using event queues. Staging has also been used for different goals. Welsh et al. [38] use staging to build Internet services that are robust to overload conditions. Larus et al. [20] enhance server performance by scheduling different stages of an application to maximize instruction and data cache locality. Click [21] is a software architecture that achieves modularity and ease of configurability in packet processing applications by defining them as a network of packet processing functions (stages) connected via channels (queues). Harizopoulos et al. [17] make a case for breaking the database system into stages to optimally exploit memory hierarchy and underlying multiprocessor support, and make the system flexible and extensible.

Staged architectures deal with the problem of allocating processing resources to the stages in several ways. StagedServer [20] uses a wavefront algorithm to allocate processors to stages. Processors, in a predefined order, execute each stage till all the work pending for the queue is complete, and then proceed to next stage. The work talks about defining thresholds that can help latency-sensitive applications, however, no details are provided. SEDA [38] employs a thread-pool controller that adds a thread to a stage when the number of requests in the queue crosses a specified threshold, and releases the thread when it is idle for a certain specified reclaim interval. The controller avoids adding too many threads to a particular stage, while exploiting the parallelism across requests in each stage.

Our work focuses on the complementary problem of determining the right thresholds and the reclaim intervals as demand for stages change over time; we determine *when* and *how many* processing resources to add and reclaim. This problem becomes non-trivial in the packet processing domain that requires stringent performance guarantees, have non-trivial overheads of reallocating processing resources, and the workloads are unpredictable at the time scale of operation. Over several years, network traffic has been shown to be bursty at various time scales and is especially hard to predict at fine time scales [25, 27, 29, 40].

Various proportionate fair scheduling algorithms [7, 16, 36] have been proposed to distribute processing resources across competing tasks (e.g., threads). These algorithms distribute resources among threads based on their weights. Our work deals with the complementary problem of determining the weights of services when the demand for the services fluctuates over time. Steere et al. [33] present a feedback-driven proportion allocator for real-rate scheduling of threads to ensure their progress; their approach is applicable to environments that do not have hard deadlines, but do have throughput requirements.

Dynamic provisioning of resources has been studied extensively in the context of web servers, clusters and shared data centers [4, 5, 8, 11, 10, 13, 24, 32, 34]. We only discuss a few of these here. Chandra et al. [8] use a combination of online measurements, and prediction based on time-series analysis, and resource allocation techniques for managing processing resources in hosting centers. Muse [11] provisions resources dynamically across applications by continuously monitoring the load and adjusting resources between competing services, so as to maximize the utility of the system. Muse has similar goals as ours in a different domain – reducing provisioning and energy in hosting centers. The main difference is that instead of evaluating the impact of a policy on application performance, we derive an adaptation algorithm from application performance constraints (e.g., a bound on packet processing delay). Urgaonkar et al. [34] show that controlled over-booking of processing and network resources in server hosting platforms can increase their utilization dramatically while providing acceptable performance guarantees to applications.

The problem of scheduling for conserving energy and resources has been explored in various other domains like multimedia servers [39], and mobile and embedded systems [12, 14, 28, 30, 37]. However, the problem has received little focus in packet processing systems. In [22], we show for various packet processing applications that the inherent fluctuations in network traffic offer significant opportunity for adapting processor allocations to reduce processor provisioning and to conserve energy. In this paper, we present an on-line algorithm for realizing these benefits.

7 Conclusion

In this paper, we present an on-line algorithm for adapting processor allocations in pipelined packet processing systems, while ensuring that the additional delay suffered by packets as a result of adaptation is deterministically bounded. The resulting Processor Allocation Algorithm (PAL) is simple, but it allocates only as many processors to stages as needed to meet packet delay guarantees, accounts for system reconfiguration overheads, and copes 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 that for a set of trace workloads PAL can reduce processor provisioning level by 30-50%, reduce energy consumption by 60-70% while increasing the average packet processing delay by less than 150μs.

Acknowledgments

We thank Aaron Kunze and Vinod Balakrishnan of Intel®Network Architecture Lab for providing us with the current implementation of the Shangri-La Resource Abstraction Layer.

References

[1] Intel IXA Software Developers Kit 3.0. http://www.intel.com/design/network/products/npfamily/sdk3.htm.

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

[3] MontaVista Software. `http://www.mvista.com`.

[4] 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.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Server. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, 2000.

[6] S. Athuraliya, V. H. Li, S. H. Low, and Q. Yin. REM: Active queue management. *IEEE Network*, 15(3):48 – 53, May/June 2001.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[8] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurement. In *Proceedings of International Workshop on QoS*, 2003.

[9] W. chang Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Transactions on Networking*, 10(4):513–528, 2002.

[10] J. Chase, L. Grit, D. Irwin, J. Moore, and S. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of International Symposium on High-Performance Distributed Computing*, 2003.

[11] 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.

[12] F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spindown Policies for Mobile Computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[13] M. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[14] J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of Symposium on Operating Systems Principles*, December 1999.

[15] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[16] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, pages 107–121, 1996.

[17] S. Harizopoulos and A. Ailamak. A Case for Staged Database Systems. In *Proceedings of 1st Conference on Innovative Data Systems Research*, 2003.

[18] J. Hu and D. Schmidt. Jaws: A framework for high performance web servers, 1999.

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

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

[21] 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.

[22] R. Kokku, T. Riche, A. Kunze, J. Mudigonda, J. Jason, and H. Vin. A Case for Run-time Adaptation in Packet Processing Systems. In *Hotnets II*, November 2003.

[23] 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*, Special Issue on Software for Network Processors, 2004.

[24] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy Management for Commercial Servers. *IEEE Computer*, pages 39–48, 2003.

[25] W. Leland, M. Taqq, W. Willinger, and D. Wilson. On the self-similar nature of Ethernet traffic. In *Proceedings of the ACM SIGCOMM*, 1993.

[26] NLANR Network Traffic Packet Header Traces. http://pma.nlanr.net/Traces/.

[27] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[28] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, pages 89–102. ACM Press, 2001.

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

[30] D. Ramanathan, S. Irani, and R. Gupta. Latency effects of system level power management algorithms. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, 2000.

[31] 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.

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

[33] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, pages 145–158, 1999.

[34] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[35] H. Vin, J. Mudigonda, J. Jason, E. J. Johnson, R. Ju, A. Kunze, and R. Lian. A Programming Environment for Packet-processing Systems: Design Considerations. In *Proceedings of Third Workshop on Network Processors and Applications (NP3)*, February 14-15 2004.

[36] C. Waldspurger and W. Wiehl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[37] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[38] 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.

[39] Q. Wu, M. Pedram, and Q. Qiu. Dynamic Power Management in a Mobile Multimedia System with Guaranteed Quality-of-Service. In *Proceedings of the Design Automation Conference*, pages 701–707, 2001.

[40] 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.