Experimental Evaluation of Load Balancers in Packet Processing Systems*

Taylor L. Riché, Jayaram Mudigonda, and Harrick M. Vin
Laboratory for Advanced Systems Research (LASR)
Department of Computer Sciences
The University of Texas at Austin

TECHNICAL REPORT: UTCS TR-04-33
(A version of this report appeared in BEACON-I, October, 2004)

Abstract:

The load balancer is a fundamental building block for implementing high-throughput applications on multi-core architectures (e.g., network processors). In this paper, we consider two canonical load balancing schemes in the context of packet processing systems: (1) *packet-level* load balancing that determines the mapping of a packet to processor independently for each packet; and (2) *flow-level* load balancing that maps a *flow* to a processor and directs all subsequent packets of that flow to the mapped processor. By identifying the application, system, and trace characteristics that affect their relative performance, we address the fundamental question: under what operating conditions, should one choose packet-level load balancing over flow-level load balancing, and vice versa?

1 Introduction

Packet processing systems are designed to process network packets efficiently. The design of these systems is governed by two requirements: (1) support high-bandwidth links (and hence, high packet processing throughput); and (2) support an ever-increasing set of complex packet processing applications (e.g., protocol conversion, firewall, Secure Socket Layer (SSL), intrusion detection, and virus scanning). The confluence of these two requirements yields scenarios where the time to process a packet exceeds the inter-arrival time of packets at a system. To address this challenge, modern packet processing systems (and in particular, network processors (NP)) utilize multi-threaded, multi-processor architectures [3]. NP architectures fore-shadow a more general trend towards the design of multi-core, multi-threaded architectures targeted for high-throughput computing environments.

Load balancer is a fundamental software building block for implementing high-throughput applications using such multi-core architectures. A load balancer attempts to scale the throughput of a system with increase in the available processors. A load balancer can balance load across processors executing replicas of a pipeline stage (in systems that partition applications into pipeline stages) or replicas of the entire applications. The two canonical load-balancing schemes are:

- *Packet-level* load balancing that determines the mapping of a packet to a processor independently for each packet. A round-robin or a weighted round-robin distribution of packets to processors are examples of packet-level load balancing schemes.
- *Flow-level* load balancing that maps a *flow*—a sequence of related packets—to a processor and directs all subsequent packets of that flow to the mapped processor.

Selecting between these two (and other related) schemes is tricky because of at least two reasons.

On the one hand, packet-level load balancing schemes can exploit greater-degree of concurrency and can achieve finer-grain distribution of workload across processors than flow-level schemes. This is because of two reasons. First, by ensuring that no more than one packet of a flow is processed concurrently, flow-level load balancing restricts the available parallelism to the number of simultaneously active flows; packet-level load balancing, on the other hand, can process in parallel all of the packets simultaneously present in the system. Second, the set of flows arriving at a packet processing system may differ significantly from each other in their characteristics (e.g., the duration for which a flow is active, the number and the rate of packet arrivals for a flow, etc.). A flow-level load balancing scheme is often unaware of such *non-uniformity* in flow characteristics when the first packet of the flow arrives. Consequently, a flow-level load balancing scheme could map, for instance, multiple *elephant* flows onto a processor and multiple *mice* flows onto another processor [6]; the resulting imbalance wastes processor resources, and increases the processor provisioning requirement to achieve a desired level of throughput.

On the other hand, packet-level load balancing schemes can incur significantly higher overhead for accessing and updating *per-flow state* maintained by most packet processing applications (e.g., the per-flow start- and finish-tags maintained by fair scheduling algorithms [7]). This is because, with packet-level load balancing, the shared flow-state may be accessed and updated concurrently by multiple processors; this leads to two sources of overhead. First, to guarantee correctness, access to flow-state must be synchronized (e.g., by using locks); the penalty to acquire a lock and the time a processor may block waiting to acquire a lock affects packet processing throughput.

*This research is supported in part by the National Science Foundation (ITR grant ANI-0326001), State of Texas – Advanced Technology Program, and Intel.

Second, in this setting, shared per-flow state is often accessed from memory-levels shared across processors; the latency for doing so is generally significantly higher than accessing per-flow state from memory that is local to each processor. This increases the total time required to process each packet, which, in turn, increases the processor provisioning required by the packet-level load balancing scheme for achieving a desired level of throughput. Flow-level load balancing schemes localize per-flow state in memory-levels local to a single processor, and thereby eliminate most of this overhead.

Although the topic of balancing load across multiple processors has received considerable attention in the literature, much of the prior work is focused on techniques for improving the efficiency of load balancing schemes (e.g., by selecting a processor based on current load on processors [4], or dynamically re-balancing the load by migrating existing flows across processors [10]). Surprisingly, little is known about how one can select from among the set of alternatives a scheme that is best suited for a particular environment.

In this paper, we address the following fundamental question: *under what operating conditions, should one choose packet-level load balancing over flow-level load balancing, and vice versa?* We define metrics for evaluating relative performance of load balancing schemes and describe our experimental methodology. We identify the application, system, and trace characteristics that affect the relative performance of packet-level and flow-level load balancing schemes.

Our results provide guidelines for selecting a load balancer. Using these guidelines, one can automate a key aspect of implementing high-throughput applications onto highly parallel architectures. This will eliminate the burden of choosing at design time a particular load balancing scheme, and thereby simplify the programmability of these systems (the primary goal of our overall research effort [21]).

The rest of the paper is organized as follows. In Section 2, we present our simulation methodology and the metrics of our evaluation. In Section 3, we describe the setup and the results of our experimental evaluation. Section 4 discusses related work, and finally, Section 5 summarizes our findings.

2 Methodology

The performance of packet-level and flow-level load balancing schemes depend upon three categories of parameters:

- *System characteristics*: The latency in accessing global (i.e., shared across processors) and local (non-shared) memory-levels. The greater is the overhead in accessing global memory-levels, the smaller is the throughput achieved by packet-level load balancing scheme.

- *Application characteristics*: (1) The total processing requirement of a single packet (denoted by *application length*); (2) the length of critical code segment that access and manipulate shared flow-state; and (3) the flow granularity (i.e., the flow definition used by the application). Since flow-level load balancing scheme only process a single packet of a flow at a time, large application lengths and coarse flow granularities restrict the parallelism available to flow-level load balancers. On the other hand, large critical segment lengths and coarse flow granularities restrict the parallelism available to packet-level load balancers.

- *Workload characteristics*: Distributions of the (1) inter-arrival time for packets; (2) arrival-rate for flows; (3) holding time for flows; and (4) sizes (in terms of number of packets) of flows. Whereas the inter-arrival time of packets determines the parallelism available to packet-level load balancers, the combination of flow arrival-rate and holding times determine the parallelism available to flow-level load balancers. The intra-flow packet arrival rate and flow size distribution, on the other hand, captures the non-uniformity in flow characteristics and thereby controls the load-imbalance that may result from flow-level load balancing.

To quantify the impact of these characteristics on load balancing, we developed an event-driven simulator. In what follows, we first describe the design of our simulator and then define the metrics for our performance evaluation.

2.1 Simulation Model

System Model : We consider a homogeneous multi-processor system. Let $\mathcal{C}$ be the number of instructions that a processor can execute in unit time. Each processor can operate independently and in parallel. We assume that each processor is configured with a local memory (or a cache) that can be accessed with a single-cycle latency. Further, processors share a memory level with access latency of $\mathcal{M}$ cycles.

Application Model : We model an application as a sequence of executions of non-critical and critical segments (that access shared flow-state) of code.

$$\mathcal{A} = \langle \mathcal{A}_1, \widehat{\mathcal{A}}_2, \mathcal{A}_3, \widehat{\mathcal{A}}_4, ... \mathcal{A}_n \rangle$$

where $\mathcal{A}_i$ and $\widehat{\mathcal{A}}_j$, respectively, denote a non-critical and a critical code segment. We characterize the execution of each segment of application code using the pair: (c_i, m_i), where c_i and m_i, respectively, refer to the number of computational and memory access instructions executed while processing a packet. Entry into each critical code segment is protected by a lock; this guarantees that only one processor can be executing inside a critical code segment at any time. For each lock, and hence each critical code segment, we specify flow granularity; for instance, a lock with flow granularity defined as the ⟨sourceIP, destinationIP⟩ address-pair serializes the processing of packets with the same ⟨sourceIP, destinationIP⟩ address-pair.

Simulator Design : Our event-driven simulator, developed in C++ and driven by TCL scripts, consists of four components: (1) a *packet reader*, (2) a *load distributor*, (3) a set of *processors* (with the associated local and shared memory levels), and (4) a *lock manager*.

The packet reader reads packets from a trace, converts it into the simulator's internal packet format, and passes it to the

load distributor. The load distributor implements load balancing schemes. Once a packet is assigned to a processor, the processor uses the (c_i, m_i) pair specified for each application code segment to simulate packet processing. On encountering a critical code segment, the processor attempts to acquire from the lock server the lock corresponding to the flowID associated with the packet. Upon acquiring the lock, the processor continues packet processing. Upon completing the processing of a packet, the processor selects the next packet to process (if one is available); otherwise, the processor is placed on an idle queue. If the lock that a processor attempts to acquire is already occupied, then the processor is placed on a waiting queue for the lock. When the lock is released by another processor, the lock server assigns the lock to the processor at the front of the waiting queue.

2.2 Performance Metric

Given a packet processing application and a traffic trace, we evaluate the performance of a load balancing scheme in terms of the *processor provisioning* needed to meet the throughput requirement of the trace. In particular, given an application $\mathcal{A}$ and a packet trace $\mathcal{T}$, we define the processor provisioning requirement for a load balancing scheme by first defining *processor capacity* and *offered load* as follows:

- We define the *processor capacity* $\mathcal{P}(\mathcal{A},\mathcal{T},n)$ as the number of packets that can be processed by a processor within time $IAT(\mathcal{T})$, the average inter-arrival time for packets in trace $\mathcal{T}$. Formally,

$$\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}(\mathcal{A},n)} * IAT(\mathcal{T}) \quad (1)$$

  where $\mathcal{C}$ is the number of instructions that a processor can execute in unit time, and $\mathcal{W}(\mathcal{A},n)$ is the number of processor cycles requires to execute application $\mathcal{A}$ for a packet on a processor in a system with n processors.

- We define the *offered load* $O(\mathcal{T},n)$ per processor as the number of packets that arrive at a processor within time interval $IAT(\mathcal{T})$ in a system consisting of n processors. If $P\_IAT(\mathcal{T},n)$ denotes the average inter-arrival time of packets observed at a processor in a system with n processors, then we define $O(\mathcal{T},n)$ as:

$$O(\mathcal{T},n) = \frac{1}{P\_IAT(\mathcal{T},n)} * IAT(\mathcal{T}) \quad (2)$$

Given the definitions of processor capacity $\mathcal{P}(\mathcal{A},\mathcal{T},n)$ and offered load $O(\mathcal{T},n)$, we define the *processor provisioning* level (denoted by $\mathcal{N}_s(\mathcal{A},\mathcal{T})$) required by load balancing scheme s as:

$$\mathcal{N}_s(\mathcal{A},\mathcal{T}) = \min\{n | \mathcal{P}(\mathcal{A},\mathcal{T},n) > O(\mathcal{T},n)\} \quad (3)$$

Given the processor provisioning levels for load balancing schemes s_1 and s_2, we quantify the relative performance of the two schemes for application $\mathcal{A}$ and trace $\mathcal{T}$ in terms of *provisioning ratio*, $\mathcal{R}$, as:

$$\mathcal{R}_{s_1,s_2}(\mathcal{A},\mathcal{T}) = \frac{\mathcal{N}_{s_1}(\mathcal{A},\mathcal{T})}{\mathcal{N}_{s_2}(\mathcal{A},\mathcal{T})} \quad (4)$$

3 Experimental Evaluation

In this section, we describe the setup and the results of our experimental evaluation.

3.1 Experimental Setup

We quantify the relative performance of load balancing schemes under the following parameter settings.

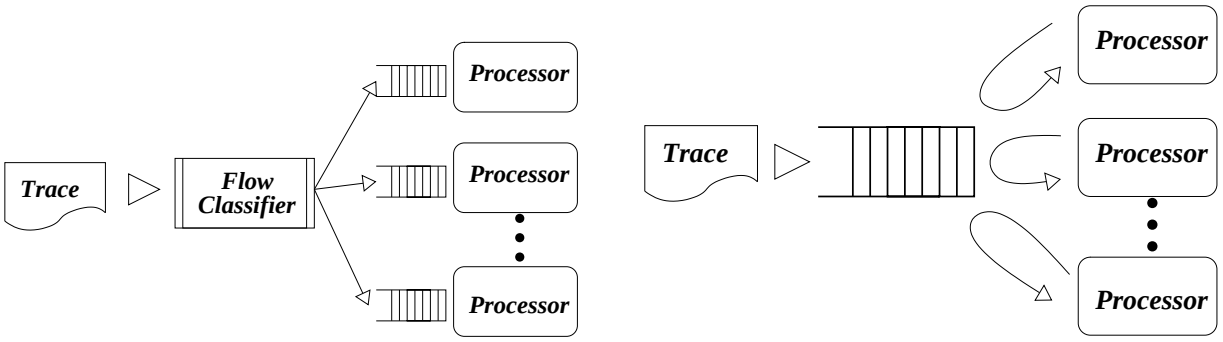
- **System characteristics**: We consider a multi-core system (e.g., Intel's IXP2800 [8]). We assume that memory local to a processor can be accessed in a single cycle; further, accesses to shared memory level requires 10, 50, 100, or 200 cycles. These values are representative of today's architectures with on-chip and off-chip shared memory levels [8].

- **Application characteristics**: We consider synthetic packet processing applications with a single critical section[1] (namely, $\mathcal{A} = \langle \mathcal{A}_1, \widehat{\mathcal{A}}_2, \mathcal{A}_3 \rangle$), with the total packet processing times selected as in integral (2, 4, 8, 16, and 24) multiple of the average inter-arrival time (IAT) of the packet trace. For each of these cases, we consider applications where the critical code segment (namely, $(c_2 + m_2)$) accounts for 1.25%, 2.5%, 5%, 10%, and 20% of the trace IAT. We consider three possible values for flow granularity: (1) 5-tuple (defined by ⟨sourceIP, sourcePort, destinationIP, destinationPort, protocolID⟩); (2) ⟨sourceIP, destinationIP⟩ address-pair; and (3) the ⟨destinationIP⟩ field.

- **Traces**: To compare load balancing schemes under realistic workloads (i.e., representative packet- and flow-level parallelism), we use traces collected from various locations in the Internet to drive our simulations. In this paper, we report results obtained using two traces: (1) the UNC traces [17] collected from the link connecting the University of North Carolina at Chapel Hill to the Internet; and (2) the MRA traces [14] collected from an OC-12 (620 Mbits/second) link that is closer to the Internet backbone. The UNC traces represent traffic pattern at the edge of the network, while the MRA traces represent traffic in the network core.

- **Load balancing schemes**: We study two practical load balancing schemes—packet-level load balancing and flow-level load balancing (as seen in Figure 1)—as well as a *hypothetical* load balancing scheme. Recall that the packet-level load balancing scheme can exploit packet-level parallelism available in a trace; however, its performance is limited by the overhead incurred in accessing flow-state from shared memory-levels as well as the serialized execution of critical code segments. Flow-level load balancing, on the other hand, processes only one packet of a flow at a time (and thereby eliminates locking) and minimizes the memory access latencies (by using memory local to a processor to maintain flow-state);

[1] In this paper, we assume this simplified application model; exploration of more complex applications with multiple critical segments is a topic for future research.



(a) Flow-level load balancing (b) Packet-level load balancing

Figure 1: Diagrams of the two load balancing schemes.

however, it can only exploit flow-level parallelism. The *hypothetical* scheme combines the best of the packet-level and flow-level load balancing schemes—it exploits packet-level parallelism and yet does not incur any memory access or synchronization overhead.

3.2 Results

In this section, we address the question: *under what operating conditions, should one choose packet-level load balancing over flow-level load balancing, and vice versa?* We first identify the application, system, and trace characteristics that affect packet processing time $\mathcal{W}(\mathcal{A},n)$, processor capacity $\mathcal{P}(\mathcal{A},\mathcal{T},n)$, and offered per-processor load $\mathcal{O}(\mathcal{T},n)$. Then, we study the impact of these characteristics on the processor provisioning ratio $\mathcal{R}_{f,p}$.

3.2.1 Packet Processing Times $\mathcal{W}(\mathcal{A},n)$

Let $\mathcal{W}_p(\mathcal{A},n)$ and $\mathcal{W}_f(\mathcal{A},n)$, respectively, denote the number of processor cycles required to process a packet in a system with packet-level and flow-level load balancing.

We assume that a flow-level load balancing scheme enables *all* memory accesses made while processing a packet to be serviced from the local memory available at a processor. Hence, for $\mathcal{A} = \langle \mathcal{A}_1, \widehat{\mathcal{A}}_2, \mathcal{A}_3 \rangle$, we get:

$$\mathcal{W}_f(\mathcal{A},n) = (c_1+m_1)+(c_2+m_2)+(c_3+m_3) \qquad (5)$$

Observe that $\mathcal{W}_f(\mathcal{A},n)$ is independent of n.

In the case of packet-level load balancing scheme, accesses to shared per-flow state (made during the execution of critical segment $\widehat{\mathcal{A}}_2$) are serviced from shared memory level (with access penalty of $\mathcal{M}$ cycles); we assume that all other memory accesses can be serviced from local memory. Further, the entry to the critical code segment is serialized using locks; let $\mathcal{L}(\mathcal{A},n)$ denote the time for which a processor may block waiting to acquire a lock to enter critical code segment $\widehat{\mathcal{A}}_2$. Hence, we get:

$$\mathcal{W}_p(\mathcal{A},n) = (c_1+m_1)+(c_2+m_2*\mathcal{M})+\mathcal{L}(\mathcal{A},n)+(c_3+m_3) \qquad (6)$$

where $(c_2+m_2*\mathcal{M})$ denotes the number of cycles required to execute critical code segment $\widehat{\mathcal{A}}_2$. Thus, to derive $\mathcal{W}_p(\mathcal{A},n)$, we need to first derive the lock waiting time function $\mathcal{L}(\mathcal{A},n)$.

Observe that $\mathcal{L}(\mathcal{A},n)$ depends on the effective length of the critical segment ($ECS = (c_2+m_2*\mathcal{M})$) and the average packet inter-arrival time for the trace. In Figure 2, we plot the variation in $\mathcal{L}(\mathcal{A},n)$ as a function of effective critical segment length. We plot $\mathcal{L}(\mathcal{A},n)$ and ECS as multiples of the trace inter-arrival time $IAT(\mathcal{T})$. Using Figure 2, we derive the following conclusion.

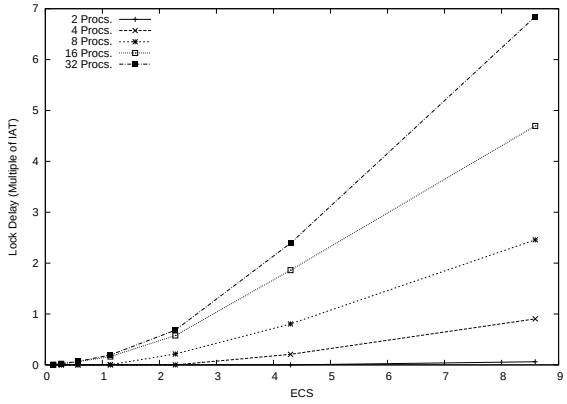


Figure 2: Variation in lock waiting time $\mathcal{L}(\mathcal{A},n)$ with n and effective critical segment length (ECS) (represented as a multiple of trace IAT).

Conclusion 1. *The lock waiting time $\mathcal{L}(\mathcal{A},n)$ increases with increase in ECS and n.*

The magnitude of $\mathcal{L}(\mathcal{A},n)$, however, is governed by the burstiness *of packet arrivals within a flow. The greater the burstiness (i.e., number of packet arrivals of a flow within interval ECS), the greater is the average lock waiting time $\mathcal{L}(\mathcal{A},n)$. For the traces we examine, only a relatively small number of packets belonging to a flow arrive within a time window ECS (see Figure 3). Hence, the observed lock waiting time is often smaller than ECS.*

3.2.2 Processor Capacity $\mathcal{P}(\mathcal{A},\mathcal{T},n)$

Recall from Equation (1) that *processor capacity* $\mathcal{P}(\mathcal{A},\mathcal{T},n)$ is defined as:

$$\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}(\mathcal{A},n)} * IAT(\mathcal{T})$$

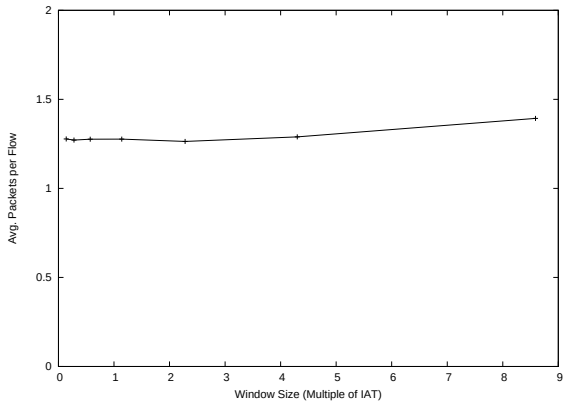


Figure 3: Intra-flow concurrency in UNC trace.

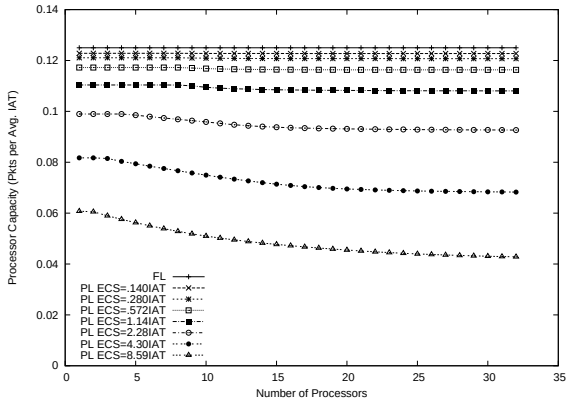


Figure 4: Processor capacity as a function of *n*.

Figure 4 plots the processor capacities obtained by substituting the values of $\mathcal{W}(\mathcal{A},n)$ for flow- and packet-level load balancing schemes derived in Section 3.2.1. We derive the following conclusion.

Conclusion 2. *For the flow-level load balancing scheme, $\mathcal{W}_f(\mathcal{A},n)$ is independent of n; hence, for a given application $\mathcal{A}$ and packet trace $\mathcal{T}$, the* processor capacity *$\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$ is independent of n, the number of processors in the system.*

On the other hand, for the packet-level load balancing scheme, since $\mathcal{L}(\mathcal{T},n)$ and hence $\mathcal{W}_p(\mathcal{T},n)$ increase with n, the processor capacity $\mathcal{P}_p(\mathcal{A},\mathcal{T},n)$ decreases with increase in n. Observe that the rate of increase in $\mathcal{L}(\mathcal{T},n)$ with n reduces significantly at large values of n; hence, the processor capacity $\mathcal{P}_p(\mathcal{A},\mathcal{T},n)$ stabilizes for larger values of n.

3.2.3 Offered Load $O(\mathcal{T},n)$

A packet-level load balancing scheme distributes packets evenly across the available processors; hence, for a system with *n* processors:

$$P\_IAT_p(\mathcal{T},n) = n * IAT(\mathcal{T}) \tag{7}$$

On the other hand, a flow-level load balancing scheme distributes load across processors in terms of flows. In the presence of non-uniform flows, packet arrivals are not distributed uniformly across the processors. Hence, we define the $P\_IAT_f(\mathcal{T},n)$ as:

$$P\_IAT_f(n) = \min_{i\in[1,n]} P\_IAT_i(\mathcal{T}) \tag{8}$$

where $P\_IAT_i(\mathcal{T})$ is the observed inter-arrival time of packets at processor *i* with packet trace $\mathcal{T}$.

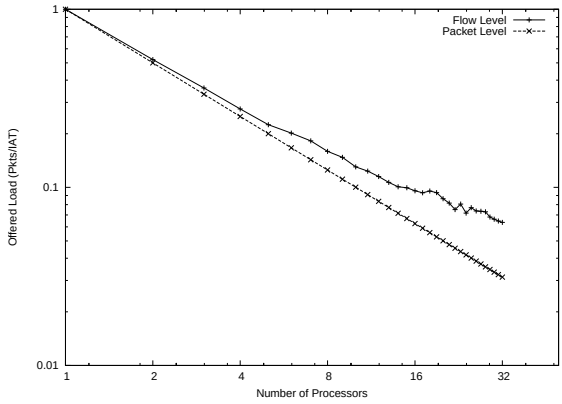


Figure 5: Offered load $O(\mathcal{T},n)$ as a function of *n*.

Using the definitions of *P_IAT* (Equations 7 and 8) and offered load *O* (Equation 2), we experimentally derive the variation of $O(\mathcal{T},n)$, the per-processor offered load, as a function of *n* for both flow- and packet-level load balancing schemes (see Figure 5). We derive the following conclusion.

Conclusion 3. *For the packet-level load balancing scheme, the offered load $O(\mathcal{T},n)$ for a processor reduces linearly with increase in n, the number of processors in the system. For the flow-level load balancer, because of the non-uniformity in flow characteristics, the reduction in the per-processor offered load with increase in n is sub-linear.*

3.2.4 Processor Provisioning Ratio $\mathcal{R}_{f,p}$

Figure 6 shows the provisioning ratio $\mathcal{R}_{f,p}$ as a function of application characteristic (namely, the effective critical segment length (ECS)) for several different processor capacity values $\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$. Observe that

$$\mathcal{R}_{f,p}(\mathcal{A},\mathcal{T}) = \frac{\mathcal{N}_f(\mathcal{A},\mathcal{T})}{\mathcal{N}_p(\mathcal{A},\mathcal{T})}$$

Hence, $\mathcal{R}_{f,p}(\mathcal{A},\mathcal{T}) > 1$ indicates that to meet the throughput demands of application $\mathcal{A}$ and packet trace $\mathcal{T}$ the flow-level load balancer requires a greater number of processors than the packet-level load balancer. Similarly, $\mathcal{R}_{f,p}(\mathcal{A},\mathcal{T}) < 1$ indicates that the flow-level load balancer requires a smaller number of processors than the packet-level load balancer. We refer to the ECS value at which the provisioning requirement of the flow-level load balancer becomes smaller than the provisioning requirement for the packet-level scheme as the *cross-over point*. We derive the following conclusion from Figure 6.

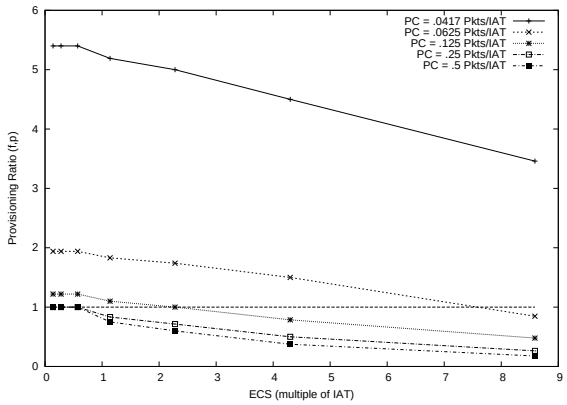


Figure 6: $\mathcal{R}_{f,p}$ for different applications $\mathcal{A}$ (UNC trace). Processor capacity (PC): $\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}_f(\mathcal{A},n)} * IAT(\mathcal{T})$

Conclusion 4. *As the value of processor capacity $\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$ decreases, the cross-over point occurs at larger values of ECS. This indicates that the packet-level load balancer remains preferable for a greater region of the application design space (as identified by the effective critical segment length).*

3.2.5 Effect of Packet Trace $\mathcal{T}$

In the previous sections, we showed the results obtained using the UNC trace. Figure 7 shows the provisioning ratio $\mathcal{R}_{f,p}$ as function of application characteristic (namely, the effective critical segment length (ECS)) for several different processor capacity values $\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$ for the MRA trace. The trends seen in Figure 7 are similar to those observed in Figure 6. However, for the same value of $\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$, the cross-over point occurs at smaller values of ECS length, indicating that in case of the MRA trace, the flow-level load-balancer becomes preferable at a smaller effective critical segment lengths than with the UNC trace.

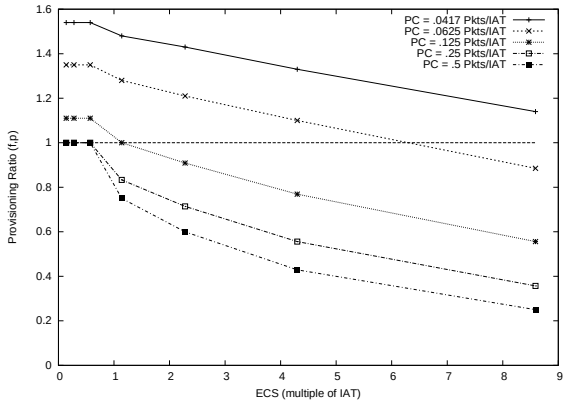


Figure 7: $\mathcal{R}_{f,p}$ for different applications $\mathcal{A}$ (MRA trace). Processor capacity (PC): $\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}_f(\mathcal{A},n)} * IAT(\mathcal{T})$

This is because, the MRA trace contains flows with similar characteristics as compared to the flows in the UNC trace. Table 1 shows the mean, standard deviation and coefficient of variation of the average inter-arrival times for packets within a flow for the UNC and MRA traces. The higher coefficient of variation for the UNC trace is indicative of a greater amount of non-uniformity in flow characteristics.

Trace	Mean	Std.Dev	Coeff. of Var.	# Flows
MRA	572675	711545	1.24	13548
UNC	699895	1593082	2.28	9393

Table 1: Flow statistics for the MRA and UNC traces. (IATs are measured in micro-seconds).

Conclusion 5. *The higher the degree of similarity in flow characteristics, the greater is the operating region over which the flow-level load balancer is preferred over the packet-level load balancer.*

3.2.6 Effect of Flow Granularity

The offered load and the lock waiting time depend on the flow granularity associated with critical segment $\widehat{\mathcal{A}}_2$. Figure 8 shows the variation in the provisioning ratio $\mathcal{R}_{f,p}$ for different flow granularities for various combinations of processor capacity $\mathcal{P}_f(\mathcal{A},\mathcal{T},n)$ and effective critical segment lengths (ECS). We draw the following conclusion.

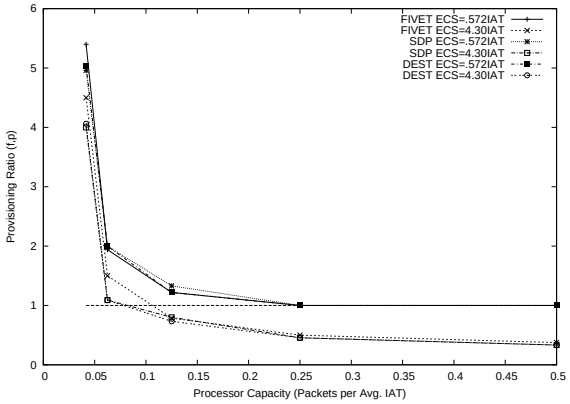


Figure 8: Variation in $\mathcal{R}_{f,p}$ for different flow granularities. Processor capacity: $\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}_f(\mathcal{A},n)} * IAT(\mathcal{T})$.

Conclusion 6. *Selecting coarse-granularity flow definitions (e.g., using $\langle$sourceIP, destinationIP$\rangle$ address-pair rather than 5-tuple) does change the provisioning ratio. This is because, with coarse-granularity flows, the lock waiting time $\mathcal{L}$ increases; this affects the packet-level load balancer. On the other hand, coarse-granularity flow definitions adversely affects the ability of the flow-level scheme to balance load across processors.*

However, the magnitude of the change is small; hence, the cross-over point between the packet-level and the flow-level load balancer does not change appreciably for different flow granularities.

3.2.7 Comparison with Hypothetical Load Balancer

In Figure 9, we compare the processor provisioning required by the two practical load balancing schemes (namely, packet- and flow-level load balancing) with that required by a hypothetical load balancer described in Section 3. We derive the following conclusions.

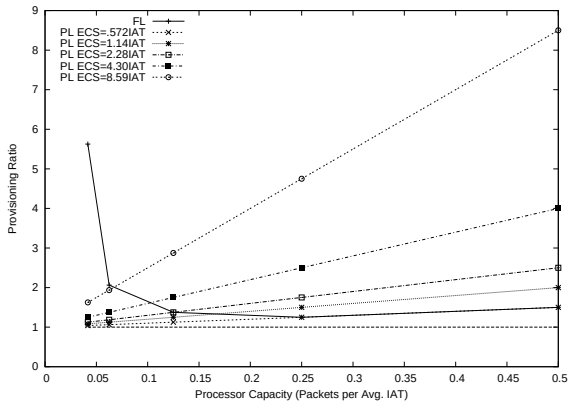


Figure 9: Variation in $\mathcal{R}_{p,h}$ and $\mathcal{R}_{f,h}$ with processor capacity $\mathcal{P}(\mathcal{A},\mathcal{T},n) = \frac{\mathcal{C}}{\mathcal{W}_f(\mathcal{A},n)} * IAT(\mathcal{T})$.

Conclusion 7. *For small processor capacities (i.e., when $\mathcal{W}_f(\mathcal{A},n) \gg IAT(\mathcal{T})$), the provisioning required by the packet-level load balancer is similar to that of the hypothetical load balancer. The processor provisioning ratio $\mathcal{R}_{p,h}$ depends upon the effective critical segment (ECS) length. For small values of ECS, the application processing time is dominated by the non-critical segments; hence, $\mathcal{R}_{p,h} \rightarrow 1$. For large processor capacities (i.e., when $\mathcal{W}_f(\mathcal{A},n) \rightarrow IAT(\mathcal{T})$), the selected ECS lengths dominate the total packet processing time. Hence, the packet-level load balancer requires a much greater amount of processor provisioning as compared to the hypothetical scheme.*

The performance of the flow-level scheme is exactly opposite of the packet-level scheme. The flow-level load balancer performs poorly at small processor capacities (i.e., when $\mathcal{W}_f(\mathcal{A},n) \gg IAT(\mathcal{T})$), but approaches the performance of the hypothetical scheme at large processor capacities. This is because, at small processor capacities, a processor can service only a small number of flows. In this case, the inherent non-uniformity in flow characteristics results in a large processor provisioning requirement. At larger processing capacities, a larger number of flows are mapped onto each processor, and hence the flow-level load balancer has greater opportunity to average-out non-uniformity in flow characteristics mapped at each processor (i.e., achieve statistical multiplexing benefits).

4 Related Work

The problem of load balancing in its most general version—to assign a set of jobs to a given number of possibly heterogeneous processors such that some system-wide metric is optimized—is shown to be NP-complete [5]. There is a vast body of literature on different heuristic-based schemes for balancing load across multiple resources. However, they do not address the two more fundamental questions. First, which scheme is well-suited in what environment? Secondly, what parameters affect the relative merits of these schemes? In this paper, we address these questions in the context of packet processing systems. For completeness, we provide a brief overview of load balancing research in various areas.

In traditional parallel computing, load balancing takes the form of scheduling a given set of (often inter-dependent) tasks on a multi-processor such that the overall execution time is minimized [16]. In general-purpose computing, since processes with widely varying processing requirements come and go, it is necessary to re-balance load across servers over time. Hence, techniques for adapting load distribution—using techniques such as process migration and affinity-based scheduling to improve the cache hit-rates—have received considerable attention [4, 18, 20]

Load balancing in web server clusters has also been an active area of research [9]. The HRW scheme [15, 19] is a popular load distribution scheme and is used in commercial products [1]. A randomized load balancing algorithm well-suited for content distribution networks (e.g., Akamai) is presented in [11]. Unlike the HRW scheme, this algorithm is analyzed under adversarial traffic conditions and has been shown to minimize the number of servers in the cluster.

Effect of shared data and the consequent locking on packet processing performance in end-system environments is studied in [2, 13]. A packet distribution scheme that aims to increase instruction cache locality is presented in [22]. An adaptive version of HRW is proposed to address the flow re-pinning problem in [10]. Special hardware that employs speculative multi-threading to efficiently handle shared data accesses is described in [12].

5 Conclusions

Load balancer is a fundamental software building block for implementing high-throughput applications using multi-core architectures. In this paper, we consider two canonical load balancing schemes in the context of packet processing systems: (1) *packet-level* load balancing that determines the mapping of a packet to processor independently for each packet; and (2) *flow-level* load balancing that maps a *flow* to a processor and directs all subsequent packets of that flow to the mapped processor. We address the fundamental question: *under what operating conditions, should one choose packet-level load balancing over flow-level load balancing, and vice versa?* We identify application, system, and trace characteristics that affect the relative performance of packet-level and flow-level load balancing schemes.

References

[1] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Comm. Magazine*, 38(5):178–184, 2000.

[2] M. Bjorkman and P. Gunningberg. Locking Effects in Multiprocessor Implementations of Protocols. In *Proceedings of ACM SIGCOMM'93*, September 1993.

[3] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall, ISBN 0-13-141792-4, 2002.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, 1995.

[5] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[6] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement*, pages 75–80. ACM Press, 2001.

[7] P. Goyal, H. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, August 1996.

[8] *Intel IXP2800 Hw. Ref. Manual*, Nov 2002.

[9] A. Iyengar, E. Nahum, A. Shaikh, and R. Tewari. Enhancing web performance. In *Proceedings of the 2002 IFIP World Computer Congress (Communication Systems: State of the Art)*, 2002.

[10] L. Kencl and J.-Y. L. Boudec. Adpative load sharing for network processors. In *Proceedings of the 21st Annual IEEE conference of Communications Society, INFOCOM 2002*, 2002.

[11] R. Kleinberg and T. Leighton. Consistent load balancing via spread minimization. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 565–574. ACM Press, 2003.

[12] S. Melvin and Y. Patt. Handling of packet dependencies: a critical issue for highly parallel network processors. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 202–209. ACM Press, 2002.

[13] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

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

[15] K. W. Ross. Hash-routing for collections of shared Web caches. *IEEE Network Magazine*, 1997.

[16] B. A. Shirazi, K. M. Kavi, and A. R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[17] 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.

[18] M. S. Squiillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, 1993.

[19] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[20] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 26–40. ACM Press, 1991.

[21] H. M. 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 the Third Workshop on Network Processors & Applications - NP3 Held in conjunction with The 10th International Symposium on High-Performance Computer Architecture*, 2004.

[22] T. Wolf and M. A. Franklin. Locality-aware Predictive Scheduling of Network Processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, Nov 2001.