

# Network Processor Load Balancing for High-Speed Links

Gero Dittmann and Andreas Herkersdorf

IBM Research, Zurich Research Laboratory  
Säumerstrasse 4, 8803 Rüschlikon, Switzerland  
{ged, anh}@zurich.ibm.com

**Keywords:** Network processor, load balancing, WAN, scalability, high speed.

## Abstract

While transmission rates already achieve speeds beyond 40 Gb/s, today's network processors are only slowly approaching 10 Gb/s. In this paper we present a load-balancing scheme that enables system designers to bridge the performance gap using multiple slower NPs in parallel to serve high-speed links. The proposed scheme works in a flow-preserving manner to ensure in-sequence packet delivery as well as local validity of connection state information, while avoiding inter-processor communication. The effectiveness of the algorithms is evaluated by simulation with extrapolated workloads, and the impact of specific parameters on system performance is the subject of a factor-relevance analysis.

## 1 INTRODUCTION

The fastest network processors (NPs) on the market today claim to be able to handle up to 2.5 Gb/s of aggregate throughput. Products for 10 Gb/s have been announced. At the same time, data rates of up to 40 Gb/s on a single wavelength in a fiber are available, and even faster systems have been demonstrated in labs [1].

This performance gap must be bridged if high-speed fibers are to be combined with the features and flexibility of NPs. Even if chip technology continues to follow Moore's Law, the improvements will not suffice to solve the problem because transmission speeds are expected to grow faster. Consequently, architectural advances are needed to scale the performance of today's NPs to the required speeds.

We present a set of algorithms that distribute packet traffic from a high-speed link across multiple lower-speed NPs in a system as shown in Figure 1. The scheme can also be applied in the egress direction—from the switch to the link. Seen from the outside, this system behaves like a single high-throughput NP with an aggregate throughput equal to the sum of throughputs of the individual NPs.

Packets that belong to the same flow are generally sent to the same NP to maintain intra-flow packet sequence and to keep flow state information local, thus avoiding the need for synchronization between NPs. The algorithms still ensure an even load distribution between the NPs. The scheme assumes

no up-front information about flow properties as is provided by resource reservations. Hence, it even applies to protocols that do not provide reservations, such as in the Internet Protocol (IP) suite. The IP suite is the focus of this work; the scheme, however, is also applicable to other protocols.

The next section introduces known concepts and the basic architecture of the load balancer. Several algorithms are presented in Section 3 that optimize the balancing decisions. Section 4 provides performance evaluation results for the system, Section 5 discusses implementation issues, and in Section 6 the paper is concluded.

## 2 BACKGROUND

### 2.1 Load-Balancer Principle

In Figure 1, two parts of the load balancer are distinguished: A *receive* (*Rx*) part, which distributes incoming packets from the high-speed link across the NPs, and a *transmit* (*Tx*) part, which re-joins the processed packets onto a single link to the switch.

Figure 2 shows the internal structure of the *receive* part. Upon entering the load balancer, a packet is first inspected

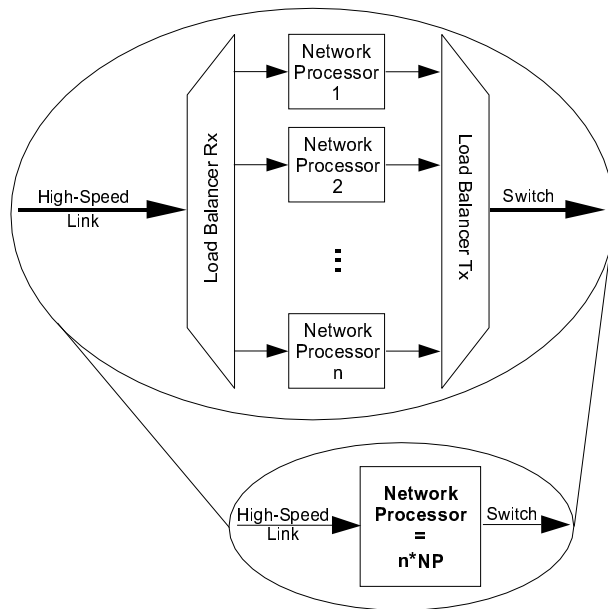


Figure 1: Load-balancer system.

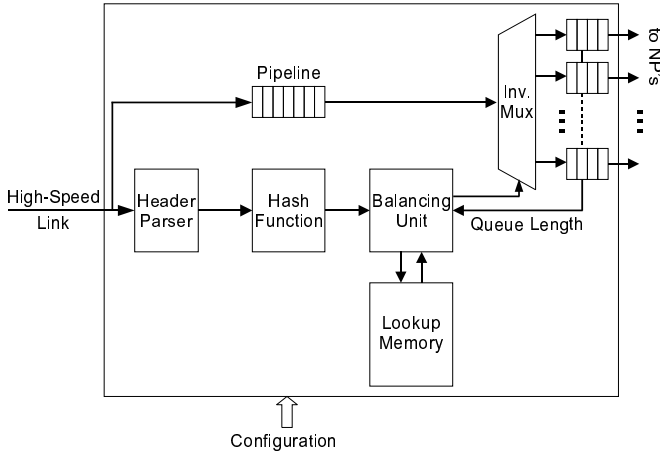


Figure 2: Load-balancer scheme.

by a header parser. Here, the fields in the packet header that uniquely identify a flow are extracted. In the case of TCP or UDP over IP for instance, this is the five-tuple consisting of source and destination addresses, both layer-4 ports, and transport protocol number. These fields are then compressed by a hash function to an index of fixed length. Even with an ideal hash function, there will be hash collisions as soon as the active flows outnumber the available hash indices. It is questionable whether resolving hash collisions is viable at link speeds of 10 Gb/s and higher, and it would undoubtedly entail increased complexity. As we will show, we achieve good balancing without this resolution. Thus, the index that the hash function delivers represents a *flow bundle*. The index serves as an address into the lookup memory of the balancing unit, where the number of one of the output queues is stored. Compared with direct hashing to a queue number, this two-stage approach allows dynamic revisions of the flow-queue assignments.

The balancing unit decides to which NP a packet will be sent based on the information stored in the lookup memory and on the length of the individual queues towards the NPs. To be implementable in the high-speed domain with today’s memory access latencies, the balancing algorithm has to be designed such that it does not require more than one read and update of a table entry per packet. For a packet size of 40 bytes (TCP acknowledgments) on a 10-Gb/s link that leaves us a memory access time per packet of less than

$$t_{\text{pkt}} = \frac{40 \text{ Bytes}}{10 \text{ Gb/s}} = 30 \text{ ns},$$

and at 40 Gb/s this is further reduced to 7.5 ns.

The balancing unit controls the inverse multiplexer, through which packets are enqueued. During the decision-making process, the corresponding packet is delayed in a pipeline until the queuing decision is available. The queues are implemented in a shared buffer to allow a certain tolerance of temporary queue imbalances. Today, memory access for target link speeds of 10 Gb/s and higher is only possible

with wide-data-bus, on-chip memories. Thus, buffer-size requirements must be constrained to enable integration on the same die.

The *transmit* part of the load balancer towards the switch in Figure 1 reunites the packet streams from the NPs. Implementation is straightforward because the NPs do not deliver more traffic than the switch link can carry and hence no congestion can occur. The NP traffic only has to be multiplexed. This can be realized with queues for speed adaptation, and a simple packet-scheduling algorithm such as deficit round robin [2].

## 2.2 Related Work

A load balancing scheme for Internet links has been described in [3]. The authors focus on evaluating hash functions for this purpose. This is important because the quality of the hash function in conjunction with the number of available hash indices is critical for the number of hash collisions. This number determines the resolution of the balancing process. We extend the concept of direct hashing by mechanisms that continuously optimize flow-bundle allocation by exploiting short flow lifetimes as well as feedback information from the queues. In this way, overload situations can be identified and relieved at an early stage.

Extensive research has been devoted to load sharing in parallel computers; see [4, 5, 6]. Some of the basic approaches found there are similar to ours: optimally distributing load across multiple processors by assigning “service requests” to less-loaded processors. Reassigning flows from one NP to another as introduced in Section 3.2 has similarities to task migration or load stealing (e.g. [6]); flows consist of packets in analogy to processes consisting of threads. There are, however, also fundamental differences:

- In a parallel computer, threads of a process usually work mainly on their private context and do not share much information with each other, i.e. they show a computation/communication ratio larger than one, often hundreds or higher. Therefore, the gain from distributing threads over several processors outweighs the communication overhead. In contrast, packets of a flow usually share *all* of their context information. With an interarrival rate of millions of packets per second, the distribution of basic applications such as policing alone would entail major communication overhead to continuously update the context between the NPs. Hence, packets of a flow generally need to be processed by the same NP.
- In a parallel computer, processors communicate with each other in order to better distribute load when an imbalance has been detected, and to share context information. For an NP load balancer it is desirable to be transparent to the NPs and to avoid the need for any communication between the NPs, as this allows using any off-the-shelf NPs without changes.

- The communication overhead for moving a thread’s context from one processor to another represents the cost of moving the thread. When this cost is lower than the benefit in the thread’s response time the balancing is executed [4].

The cost of moving a flow to another NP is a transient inconsistency of the flow’s statistics. Moreover, packet order within a flow cannot be guaranteed when packets are being processed by different NPs. But balancing might be needed to prevent buffer overflows. Here, the trade-off is a different one: service degradation vs. potentially dropping packets because of buffer overflow.

- With insufficient load sharing in a parallel computer the response time is sub-optimum. Excess threads are stalled for later execution and cannot spawn new threads in the meantime. When our load balancer does not equalize the load of the NPs sufficiently, then the shared buffer will fill up and eventually packets will have to be dropped, which is highly undesirable.
- Computing threads normally have a well-defined end. In contrast, it cannot generally be assumed that flows advertise their termination. As the termination of all flows in a bundle is the perfect moment for revising its queue association, this event needs to be detected, as described in Section 3.1.
- A thread that spawns many demanding new threads takes a long time to compute on one processor, but a flow that delivers more traffic than the maximum throughput of one NP simply cannot be handled by this NP. Moving this flow to another NP will only shift but not solve the problem. A mechanism to avoid packet drops in this situation is introduced in Section 3.3.

### 3 LOAD DISTRIBUTION SMOOTHING

In the simplest case, the only information stored in the lookup memory is the assignment to a queue for each flow bundle. This would be a completely static association between flows and NPs. Because flow characteristics, such as average data rate and burstiness, are not predictable in a reservation-less environment, these properties cannot be used for optimally setting up a static flow distribution. A random setup can lead to massive imbalances in NP occupation and consequently to huge differences in queue lengths. While one queue might be building up over time, another queue might be idling. To avoid this scenario and to optimize the balancing decision, flow dynamics must be taken into consideration. This is realized by our algorithms, which are introduced in the following.

### 3.1 Flow Time-Out

If for a longer period of time no data arrives from a particular flow, it can be assumed that this flow has terminated. If that is true for all flows in a flow bundle, the association between the flow bundle and a queue can be changed safely without negative impact on the traffic. No packet reordering can occur after the last packet, and flow statistics in the NPs expire at the end of the flow.<sup>1</sup>

This fact is exploited by storing a time stamp for each flow-bundle ID in the lookup memory, which is updated every time a packet arrives with that ID. When the next packet with the same ID comes in, the stored time stamp is compared with the actual system time. If the difference is greater than a configured time-out value, the association with an NP has expired and is replaced by the NP with the currently shortest queue (*minQ*). In this way, newly initiated flows are assigned to the least-loaded NP, which smoothes out long-term imbalances in the load distribution.

```
for each packet {
    flow ← lookup(packet.id)
    if (currentTime - flow.timeStamp > timeOut) {
        flow.targetQ ← minQ()
    }
    flow.timeStamp ← currentTime
    packet.targetQ ← flow.targetQ
}
```

### 3.2 Flow Reassignment

Although the flow time-out mechanism helps avoid overload situations for the NPs, it does not prevent them completely. Its effectiveness depends on the frequency of flow expirations, i.e. the fewer flows time out, the less smoothing takes place. Also, burstiness of traffic and sustained data-rate changes of flows over their lifetime can lead to short-term load imbalances between queues, which cannot be handled well with flow time-outs only.

The proposed solution is a variation of sender-initiated adaptive load sharing [5] with a global task queue. In our case not only single threads are migrated but entire flow bundles are redirected away from overloaded queues, because all packets of a flow share their complete context. A queue is considered to be overloaded if it occupies more than a configurable share of the shared buffer. This condition is tested for every incoming packet. If the output queue to which a packet should be sent exceeds the buffer share threshold, then the NP currently associated with that flow bundle in the lookup memory is substituted with the *minQ*. The packet and all following packets of that flow bundle are sent to the new queue. This is called *flow reassignment*.

<sup>1</sup>Even for flow bundles that time-out before they really terminate, reordering can be denied if the time-out period is adjusted to be longer than packet latency through the NP.

```

for each packet {
  flow ← lookup(packet.id)
  if (flow.targetQ.size > qThreshold) {
    flow.targetQ ← minQ()
  }
  packet.targetQ ← flow.targetQ
}

```

Unlike the parallel computer environment and thanks to the one-chip implementation with centralized scheduling, the balancing unit knows the fill levels of all queues and thus finds the minQ without relying on random probing. This is vital because the short interarrival time between packets does not allow for the probing latency. Furthermore, as the threshold will be above the size of the queue buffer divided by the number of NPs (otherwise there would be no benefit in having a *shared* buffer), there will always be a queue shorter than its share as an appropriate receiver for the reassigned flow—reassignment attempts are always successful.

The implication of this procedure on the traffic is the possibility that the first packets being sent to the new, shorter queue *may* overtake the last packets that were sent to the old, long queue before the reassignment. The statistics for that flow bundle being kept in the overloaded NP are no longer accurate, and new statistics have to be collected in the new NP. However, this occurs during a very brief transition time, after which packet order and flow state become, and stay, consistent again. The undesirable alternative would be packet drops as queues build up beyond buffer capacity.

Flow reassignments also help solve the problem of accommodating a flow that exceeds the free capacity of any individual NP although it is less than the combined free capacity of all NPs (Figure 3). This can be a consequence of an even load distribution. Now, the large flow is assigned to one of the queues. Because the old flows in that queue combined with the new one exceed the capacity of one NP, the queue starts filling up until it reaches the buffer share threshold. Then, the reassignment mechanism redirects flows from that queue until the overload is relieved.<sup>2</sup> In this way, all flows can be served.

Another side effect of reassignments is an inherent tolerance against NP failure. If one of the NPs fails, then it no longer serves its queue. Consequently, the queue builds up until the reassignment threshold is reached. As now the queue will not drop below the threshold again, all flows will be drawn away from that NP and are distributed among the other NPs. Evidently, the aggregate throughput of the system is reduced by one NP. But this portion of the traffic can be served at full quality, and the valuable high-speed link does not go down.

<sup>2</sup>If the mechanism accidentally moves the large flow to another queue, then flow reassignments start on this new queue. However, as even most “large” flows are responsible for only a small share of packets in a queue, this scenario is not very likely.

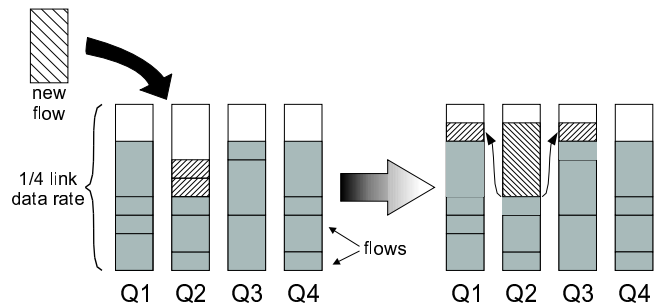


Figure 3: Large flow accommodation through reassignments.

In a similar fashion, reassignments can support outputs with different speeds, for instance due to different processing loads in the NPs. Queues towards slower outputs will grow faster than queues towards faster outputs. As the queue length is the basis for flow assignment and reassignment decisions, fewer flows will be assigned to slower links. This even works automatically, without having to configure any weights.

### 3.3 Packet Spraying

Another kind of flow bundle to be handled is one that by itself exceeds the total throughput of one NP. Such an *excessive flow* is detected using *data counters* that sum up the amount of data arriving for each flow bundle within a measurement period. A threshold on these data counters determines when a flow is considered excessive. If the measurement period corresponds to the time it takes to completely fill the buffer at full link data rate, then the threshold can be interpreted roughly as a percentage of the buffer that a single flow bundle may occupy.

At the end of the measurement period, the data counters are normally reset to zero. Even for excessive flows this would mean that the counter drops back below the threshold. This guarantees that flow bundles that return to non-excessive behavior are granted normal service again. But in order to recognize continuously excessive flows more quickly at the beginning of a period, data counters that have exceeded the threshold are only reduced to a fraction of their value instead of zero (see lower graph in Figure 4). Thus, an excessive flow has to send below the maximum throughput until it no longer exceeds the threshold before it is regarded as well behaved again. The time this takes, however, is bounded by the maximum value of the data counters.

Packets from excessive flow bundles could simply be dropped. But because the throughput of all NPs together should match the data rate of the link and thus be able to handle all packets, this seems to be an unnecessary degradation of service. An alternative is to send packets from an excessive flow bundle to the currently shortest queue. As the minQ will change over time, this leads to the packets being distributed across several NPs. We call this *packet spraying*.

```

for each packet {
  flow ← lookup(packet.id)
  packet.targetQ ← flow.targetQ
  if (currentTime - flow.resetTime > tMeasure)
  {
    if (flow.count > countThreshold) {
      flow.count ← flow.count * reduction
    } else {
      flow.count ← 0
    }
    flow.resetTime ← currentTime
  }
  if (flow.count > countThreshold) {
    packet.targetQ ← minQ()
  }
  flow.count ← flow.count + packet.size
}

```

Obviously, some packets might be reordered, but for receivers that are able to resequence packets this service is preferable to simply dropping packets. Moreover, statistics in the NPs for sprayed flows will not be consistent. If policing is an issue here, then the NPs should enforce policies earlier, well before a flow approaches the maximum throughput of an NP.

However, flow reassignments and, even more so, packet spraying are emergency mechanisms that are triggered only in rare, extreme situations. Simulation has shown that with current Internet traffic, flow time-outs occur often enough to provide an even load distribution, and they have no negative impact on service quality. With reassignments, the impact is also negligible, as argued above.

Packet spraying, on the other hand, provides a means to degrade service in a controlled manner for otherwise problematic flows. And most importantly, *only* these problematic flows will be affected, while all others are isolated and ex-

perience normal forwarding. Nevertheless, even problematic packets are forwarded.

## 4 PERFORMANCE EVALUATION

In this section, the performance of the load-balancer algorithms presented above is evaluated in a series of simulations of a system with four output queues, which is a very likely scenario. The following are important metrics to evaluate:

- Maximum buffer size.
- Number of reassignments per occupied lookup-table entry, i.e. per flow bundle.
- Percentage of sprayed packets out of all packets transmitted.

The maximum buffer size is the size that would be needed to avoid packet drops completely. A major goal is to keep this buffer size small enough to allow on-chip implementation with embedded DRAM, i.e. not larger than a few megabytes. Only on-chip memory allows the high memory bandwidth that is needed to write all data from a high-speed link *and* read it again.

The reassignment ratio gives the mean number of reassignments per flow bundle that occurred during a simulation run. Both reassignment ratio and spraying ratio should be low because of their impact on traffic.

### 4.1 Workload

For the workload, real Internet traffic traces from OC-3 (155 Mb/s) backbone links are employed, which are publicly available from the NLANR web site [7]. As described in [8], several techniques are used to scale these traces up to the desired link speeds, at full link load. Furthermore, traffic synthesis methods are used, e.g. employing self-similarity models.

Today's Internet traffic consists mostly of very short-lived low-data-rate flows. To model this kind of traffic on a fully loaded high-speed link without losing the original traffic characteristics, multiple OC-3 traces are combined. The gaps between packets are eliminated for full load and the traces are time-compressed to reach 40 Gb/s. Finally, packets from various traces are interleaved to resemble the accumulated traffic of many lower-speed links that would be found on a backbone link today. For a more detailed description of this scaling method and the resulting traffic characteristics, see [8].

For the load balancer, this kind of traffic results in flow time-outs occurring very frequently, which allows the flow distribution to be optimized continuously. Moreover, statistical multiplexing works well owing to the high number of flows. Therefore, this kind of traffic does not confront the load balancer with any challenges, even under full link load. Simulation shows that the required buffer size stays low, and only few reassignments and no sprayings occur. In fact, this is

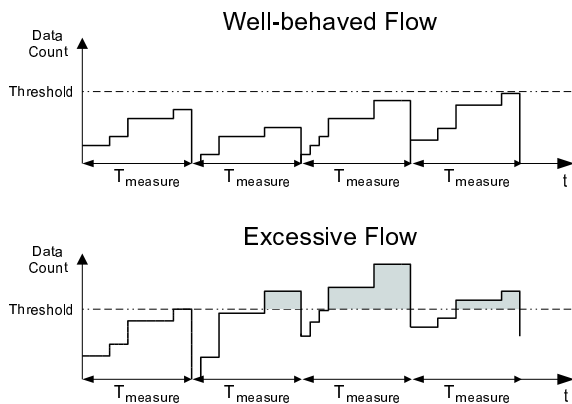


Figure 4: Data counter periods.

true already at link speeds below 40 Gb/s, and results improve with increasing link speed. Hence, the simulation results with these traces are not described in further detail here.

When trying to predict possible future traffic, we must introduce flows that are very long-lived and have data rates that at least temporarily approach and exceed the capacity of a single NP. For instance, deployment of virtual private network (VPN) tunnels will possibly entail the occurrence of such flows. These flows may provoke reassignments and spraying; buffer requirements grow. To simulate this kind of extreme situation for the load balancer, real traffic traces are time-compressed and concatenated, resulting in flows with the above-mentioned characteristics. In [8], a suite of these traffic traces is composed that offers a broad spectrum of scenarios in terms of traffic properties. All simulations below are based on this traffic suite, which represents the worst case of how we expect future traffic to be, and therefore leads to conservative results.

## 4.2 Effectiveness of the Algorithms

When confronting the load balancer with the traffic suite while only flow time-outs are enabled, i.e. with reassignments and spraying turned off, the resulting buffer sizes at times even exceed 100 MB. One simple explanation is that one flow constantly exceeds the throughput of an NP. As this flow would remain assigned to the same queue, it would persistently overflow this queue. Without reassignments, multiple flows that are assigned to one queue can overload this queue in the same fashion—with a similar result.

When reassignments and spraying are turned on, the picture changes dramatically. Figure 5 shows how buffer requirements are reduced when these algorithms are used. The thresholds at which they take effect are gradually decreased from 100% of the buffer space, which is the maximum possible value, to 25%, which corresponds to the share of one NP in the simulated four-NP system.

- In the upper graph, spraying reduces buffer size for a trace from 36.5 MB without spraying to 1.1 MB with a maximum of only two sprayed flows.

With a high threshold ( $\geq 55\%$ ), no packets are sprayed because the data counters never reach the threshold and, thus, even the largest flows are not detected as being excessive. The maximum buffer size is large (36.5 MB). For a threshold of 50% or lower, an excessive flow is detected, spraying is activated, and the maximum buffer size is reduced dramatically. The curves essentially represent step functions because of the binary property of a flow to either exceed the threshold or not. Once a flow has been perceived as being excessive, most of its packets (here between 7.7% and 9.4% of the total traffic) are sprayed.

- The lower graph shows the effect of reassignments with a different trace: the maximum buffer size is reduced

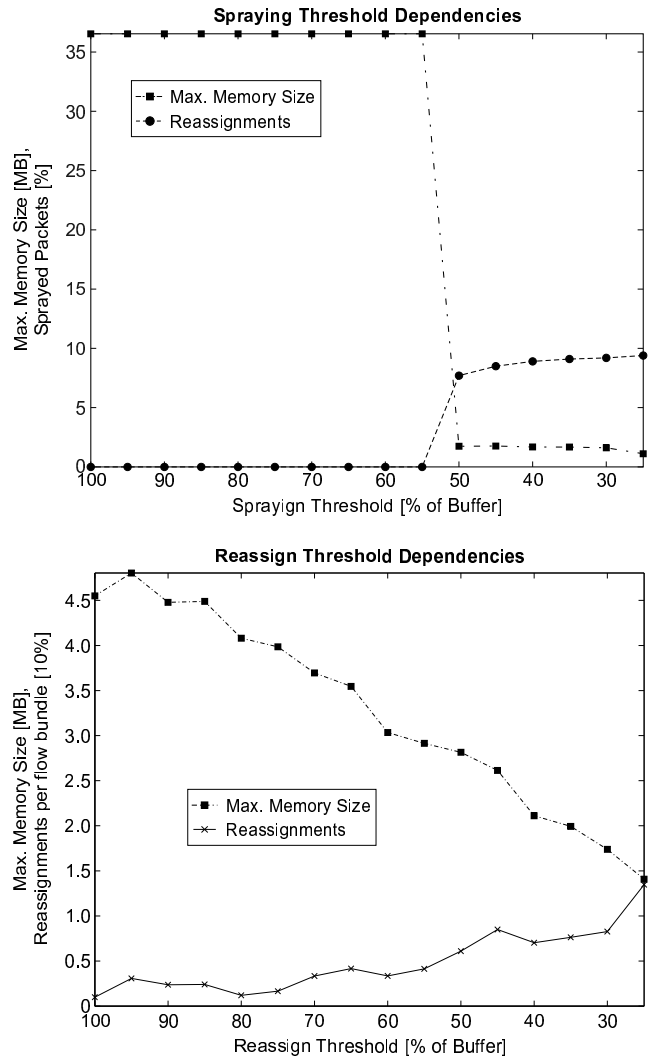


Figure 5: Effect of spraying and reassignments.

from 4.8 to 1.4 MB with reassignment ratios from 1% to 13.5% of active flow bundles (i.e. hit hash indices). Without reassignments, the buffer size grows to as much as 12.7 MB. Compared with the first graph, the curves here have a more linear character because reassignments have a smaller and less abrupt effect.

Figure 5 demonstrates the significant effect of reassignments and spraying on required buffers sizes, and how they can help keep the buffer size low enough to enable on-chip implementation, even with flows that show extreme behavior.

Simulations across the traffic suite with different threshold settings prove that for every trace the maximum buffer size can be bounded below 3 MB with moderate reassignment ratios and few sprayed flows. However, the point of lowest buffer requirement is reached with different settings for different traces. Therefore, system parameters should be tuned to the (long-term) characteristics of the actual traffic.

## 4.3 Estimation of Factor Relevance

### 4.3.1 Experimental Design

To understand the influence of various factors on system performance, a simulation for a four-NP system was carried out. The system parameters that were varied during simulation are listed in Table 1. For each factor the range in which it was varied is indicated.

Table 1: Levels for fractional factorial design.

Factor	min.	max.
Assumed buffer size	2 MB	3 MB
Flow time-out	20 ms	200 ms
Reassignment threshold	25%	75%
Spraying threshold	25%	50%
Data counter reduction	0%	80%

A *full factorial design*, in which all combinations of factors and their levels are simulated, requires a huge number of experiments. To reduce this number, a *fractional factorial simulation* was designed according to [9], in which factors take on only extreme values (see Table 1) and only a reduced set of factor-level combinations is simulated. Our design uses five factors, each alternated between two extreme values, with the number of experiments reduced to a quarter, i.e. it is a  $2^{5-2}$  design, resulting in eight experiments per simulation run. Analysis of the experiments according to [9] then yields the individual effect of factors on performance metrics as well as the combined effect of specific factors.

Test simulation runs have revealed a nonlinear dependency between flow time-out and maximum buffer requirement for flow time-out values below 20 ms. This appears to be a resonance effect with the internal time constants of the traces. Consequently, 20 ms has been chosen as the lower bound. A maximum flow time-out value of 200 ms is assumed to be sufficient even for multimedia applications.

Reassignment and spraying thresholds are given as percentages of the assumed buffer size. Values below 25% are not considered because this corresponds to the sustained performance and throughput that a single NP is expected to deliver. Data counter reduction gives the percentage to which a data counter’s value is reduced if it has exceeded the threshold in the preceding measurement period. 0% results in a counter being reset to zero.

The *assumed buffer size* factor is only needed as a reference point for the thresholds, i.e. it defines a 100%-filled buffer. It must be clearly distinguished from the *maximum buffer size* as a performance metric, which represents the size that would have been needed in a simulation run to completely avoid packet drops.

The maximum buffer size is now measured relative to the assumed buffer size, i.e. divided by either 2 MB or 3 MB. This allows buffer utilization to be compared for different as-

sumed buffer sizes. If the ratio is greater than 1, then the assumed buffer experienced an overflow, and packets had to be dropped. Other metrics are the reassignment ratio and the spraying ratio as defined earlier.

Simulation runs with different traffic traces cannot be considered simple replications of experiments because the individual traces represent completely different traffic scenarios at different points in the network that would be accommodated by different parameter settings. This view is supported by the wide spread of results. Thus, standard error analysis as used for repeated measurements in [9] is not applicable here.

Therefore, an alternative way has to be found to summarize results for different workloads. The approach chosen is to compute factor effects separately for each traffic trace. Then the mean of the effects per factor is calculated<sup>3</sup> and normalized to 100%. Only those traces are considered for which spraying and reassignments have each been involved in at least one experiment. From other traces no information about the relative importance of either mechanism can be inferred.

The results of the analysis given in Table 2 show the degree to which the individual factors, and combinations thereof, are responsible for variations in the system performance.

Table 2: Effect of factors.

	Buffer utiliz.	Reassign-ments	Spraying
Assumed buffer size	3.7%	14.9%	17.8%
Flow time-out	1.8%	1.5%	–
Reassign threshold	90.6%	61.9%	–
Assumed buffer size & flow time-out	0.6%	2.4%	–
Assumed buffer size & reassign threshold	1.1%	7.6%	–
Spray threshold	1.3%	6.8%	49.1%
Counter reduction	0.9%	4.9%	33.2%

### 4.3.2 Interpretation of Results

Owing to the above-mentioned wide spread of results, the measurement error of the effects is just as large, and the numbers only represent tendencies of factor relevance. Still it is possible to draw conclusions because the most important effects are considerably larger than others. Moreover, although the absolute numbers differ significantly between workloads, the order of the effects is essentially consistent across workloads. Bearing this in mind, we interpret the results in Table 2 as follows:

The by far most important factor for buffer utilization is the reassignment threshold. If this threshold is set low, then even small queue imbalances are smoothed out by reassignments.

<sup>3</sup>As the effects are computed as ratios, the geometric mean is used here.

Thus, queues hardly grow. Investigation of the results for individual traces showed that only for traffic with the highest spraying ratios do spraying threshold and data counter reduction also have a strong effect on buffer utilization. The earlier spraying is activated, the earlier imbalances due to excessive flows are smoothed out. The stronger effect of reassignments stems from the fact that spraying only works on the few excessive flows, whereas reassignments optimize the load distribution independently of the characteristics of individual flows.

Not surprisingly, reassignments are dominated by the setting of the reassignment threshold. Assumed buffer size also has an influence because it gives more room for queue imbalances before reassignments occur. Only for traces that are heavily sprayed is the reassignment ratio sensitive to the spraying parameters: If spraying is delayed, then reassignments try to fix the queue lengths in the meantime.

The spraying ratio does not at all depend on flow time-outs or on reassignments because these mechanisms obviously do not influence the data arrival pattern of individual flow bundles, which is the only input that is considered to activate spraying. Instead, the ratio depends on the assumed buffer size because spraying thresholds are given as a percentage of the assumed buffer size. Whether the threshold or the counter reduction dominates the spraying ratio depends strongly on the traffic characteristics. If the traffic is very bursty but does not really have high-data-rate flows, then spraying can be avoided by a high threshold. Flows with a long-term excessive data rate can profit from a low counter reduction and get quite a number of packets through that are not sprayed.

An interesting observation is the fact that flow time-out does not seem to have any significant influence on either of the performance metrics. This means that it can be set to long periods without compromising the performance of the load balancer, thereby avoiding the disruption of flows that have not yet terminated.

The system can cope with the smaller of the two assumed buffer sizes because the influence of the buffer size on system performance is not dominant for any of the three metrics. Its effect, however, is not negligible either and grows with decreasing buffer size. Thus, it seems advisable not to go much below 2 MB.

## 5 IMPLEMENTABILITY

The extraction of the flow identification out of a packet header is a relatively simple task that can be implemented in sequential logic with deterministic execution time. This statement is supported by the following back-of-the-envelope calculation: The header can be analyzed 256 bits at a time, which results in a cycle time of approximately 6 ns for 40 Gb/s line speed. With a 0.11- $\mu\text{m}$  CMOS process this allows about 40–50 levels of logic, which is sufficient even for complex header structures.

Furthermore, because resolution of hash collisions was avoided in order to enable high-speed implementation, the

subsequent hash function block is a standard exercise. Hence, these two building blocks are not problematic to implement, even for 40 Gb/s.

The balancing unit stores four values per flow bundle in its lookup memory: the associated output queue, the time stamp for the last arrived packet, the data counter, and the time stamp of the data counter reset. This information needs to be read and updated for each incoming packet. To allow a throughput of millions of packets per second, all values are stored in a single memory word, which can then be read or written with only one memory access. In this way, the aforementioned 7.5 ns per packet are sufficient.

Based on the stored data the balancing decision is made. The combinatorial logic that performs the actual decision making and updating basically consists of comparisons:

- Is the difference between time stamps and the system time greater than configured time outs?
- Is the sum of the data counter and the actual packet length greater than the configured spraying threshold?

These comparisons can be computed in parallel, and are followed by only few levels of logic to infer the balancing and updating decisions. The state information is updated and written back into the lookup memory, again with only one access. Hence, for every packet only two memory accesses are needed—one read and one write.

As analysis has shown, the requirement to use on-chip memory for the output queues can be met with a shared buffer size of around 2 MB. Combined with sophisticated memory organization such as the one found in [10], the required memory access throughput of twice the device throughput—for read and write—can be achieved.

## 6 CONCLUSIONS

In this paper we have introduced a set of algorithms for scaling NP performance by parallel use of multiple NPs. In general, the concept works in a flow-preserving manner to ensure in-sequence packet delivery and to avoid inter-NP communication. We deviate from this principle only in extreme situations when the alternative would be to drop packets. This allows the system to handle even flows that exceed the capacity of a single NP. The scheme is also tolerant of NP failure. If one NP fails, the system stays operational; the only consequence is that the aggregate throughput is reduced by one NP.

Performance analysis has shown that the algorithms reduce the required buffer size for a load-balancer solution to only 2 MB even for extreme traffic scenarios. This enables the use of on-chip memory, which is a prerequisite for reaching the memory bandwidth needed for high-speed adapters.

As no special communication is necessary between the load balancer and the attached NPs, the scheme is independent of NP vendors. The load balancer is currently being

implemented in a standard 0.11- $\mu\text{m}$  CMOS process for full OC-768 line speed (40 Gb/s).

## ACKNOWLEDGMENTS

The authors thank Patricia Sagmeister for her extensive work on synthetic and real traffic traces. We also thank Dave Webb, François Abel, and Ronald Luijten for their contributions to the concept of the Load Balancer.

## REFERENCES

- [1] K. Uchiyama and T. Morioka. “All-optical signal processing for 160 Gbit/s/channel OTDM/WDM systems.” In *Proceedings of Optical Fiber Communication Conference and Exhibit (OFC 2001)*, March 2001.
- [2] M. Shreedhar and G. Varghese. “Efficient fair queuing using deficit round robin.” In *Proceedings of ACM SIGCOMM '95*, August 1995, pp. 231–242.
- [3] Z. Cao, Z. Wang, and E. Zegura. “Performance of hashing-based schemes for Internet load balancing.” In *Proceedings of IEEE Infocom 2000*, March 2000, vol. 1, pp. 332–341.
- [4] M. Squillante and R. Nelson. “Analysis of task migration in shared-memory multiprocessor scheduling.” In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 143–155.
- [5] D.E. Eager, E.D. Lazowska, and J. Zahorjan. “A comparison of receiver-initiated and sender-initiated adaptive load sharing.” *Performance Evaluation*. 6:53–68, 1986.
- [6] M. Mitzenmacher. “Analyses of load stealing models based on differential equations.” In *Proceedings of the Tenth Annual ACP Symposium on Parallel Algorithms and Architectures*, June–July 1998, pp. 212–221.
- [7] NLANR Measurement and Operations Analysis Team. “Passive measurement and analysis.” <http://moat.nlanr.net/PMA/>.
- [8] P. Sagmeister, G. Dittmann, A. Herkersdorf, and D. Webb. “Methodology for testing high-speed network devices with predicted traffic.” Presented at *Gigabit Networking Workshop (GBN'2001)*, April 2001. <http://www.comsoc.org/tcgn/conference/gbn2001/>
- [9] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, New York, 1991.
- [10] S. Iyer and N. McKeown, “Making parallel packet switches practical.” In *Proceedings of IEEE Infocom 2001*, April 2001, pp. 1680–1687.