Packet classification in the NIC for improved SMP-based Internet servers

Éric Lemoine[†] CongDuc Pham[‡] Laurent Lefèvre[‡]

[†] Sun Microsystems Laboratories
180 Avenue de l'Europe - ZIRST de Montbonnot - 38334 St Imier - France
Eric.Lemoine@Sun.com

[‡] LIP Laboratory (UMR 5668 CNRS - ENS Lyon - UCB Lyon - INRIA) RESO team
46, allée d'Italie - 69364 Lyon Cedex 07 - France

Abstract

This document describes a new networking subsystem architecture built around a packet classifier executing in the Network Interface Card (NIC). By classifying packets in the NIC, we believe that performance, scalability, and robustness can be significantly improved on shared-memory multiprocessor Internet servers. In order to demonstrate the feasibility and the benefits of the approach, we developed a software prototype (consisting in extensions to the Linux kernel and modifications to the Myrinet NIC firmware and driver) and ran a series of experiments. The obtained results, presented therein, show the relevance of the approach.

1 Introduction

The explosive growth of the Internet, both in terms of number of users and speed of its constituents (links and routers) results in increasing absolute network loads on Internet servers[1], with large deviations around the mean network load. In addition, network speed tends to increase faster than CPU and memory speeds. For example, there is a common conception that 1 megahertz (MHz) of CPU speed is required to drive 1 megabit per second (Mbps) of network throughput. Following this paradigm, 10 gigahertz (GHz) processors are required to drive 10 gigabit per second (Gbps) network technologies (e.g. 10Gbps Ethernet [1]). At time of this writing, 10Gbps Ethernet products start being commercialized [2] and 10GHz processors are unlikely to be available anytime soon. Therefore, Internet servers must deal with increasing network loads relative to their compute power.

This gap between network and end-system speeds causes performance and robustness issues on Internet servers. Indeed, Internet servers must provide high performance for the mean network load, and delivered performance must not degrade when the offered network load exceeds their capacities. Maintaining a constant level of performance under overload is not obvious. For instance it is well known that many Unix and non-Unix based networking subsystems suffer from poor network overload behavior [3].

One way to alleviate the performance problem is *parallel network protocol processing*, i.e., using multiple processors for the execution of the protocol stack. Parallel network procotol processing has generated considerable interest in academia and industry this last decade (e.g. [4, 5, 6]). Although many approaches to parallel network protocol processing have emerged, two have gained favor: *packet-level parallelism* and *connection-level parallelism*. In packet-level parallelism, the packet is the unit of concurrency. Parallelism is achieved by dispatching packets among processing elements. In connection-level parallelism, the unit of concurrency is the connection. Parallelism is achieved by demultiplexing packets early—before they enter the network stack—to their respective connection and dispatching connections among processing elements. Packet-level parallelism allows parallelism within a single connection. With connection-level parallelism, there is no parallelism with a single connection but contentions on memory (cache lines) and locks are less likely to happen than with packet-level parallelism, thereby yielding better performance.

Networking subsystems' robustness problems have been quite studied in the past. A a few solutions to have been proposed [3, 7, 8], with some of them implemented in today's mainstream operating systems [9].

Large-scale Internet servers must deal with huge numbers of connections, thus, we believe that connection-level parallelism is the most appropriate approach to parallel network protocol processing on Internet servers. However,

[1] By Internet servers, we mean server machines connected to the Internet, therefore potentially serving a large number of clients.

building a network subsystem based on connection-level parallelism that behaves gracefully under input overload is not obvious. In this paper, we argue that classifying packets off-kernel, in the NIC, is a good solution to building efficient and robust network subsystems. We present the KNET networking subsystem built around a packet classifier in the NIC, and provide experimental results showing the relevance of our design.

The remainder of this document is outlined as follows. Section 2 and 3 formulate the problems we address in this work. More specifically, Section 2 discusses performance issues, and Section 3 discusses robustness issues. Section 4 describes our prototype's design and implementation in details. In Section 5 , we describe the software and hardware environments used for our measurements, then present the obtained results along with an analysis of these results. Finally we provide conclusions and future work in Section 6.

2 Performance issues

With the increasing number of Internet users, there is huge demand for high bandwidth networks. However, as network bandwidth increases, CPU and memory systems of Internet servers become bottlenecks, making it impossible for an Internet server to deliver the maximum bandwidth, i.e., that of the underlying network medium. One solution to this problem is to have multiple processors that simultaneously transfer data towards the network.

In this section, we begin by giving a quick overview of the working of a Internet server on a Unix-based platform in order to highlight issues relative to performing simultaneous transfers. We then present a simple model allowing to predict when having multiple processors simultaneously executing the network stack can lead to performance gains.

Internet servers typically use the HTTP protocol [10, 11], itself layered atop the TCP protocol [12]. From now on, we will use the terms Internet servers and HTTP servers interchangeably. Communications between Internet clients and servers follow the client/server model: upon reception of an HTTP request from a client, an HTTP server parses the request, forms the HTTP response[2] and initiates the transfer of the response towards the client by calling operating system's transmit primitives (e.g. `sendmsg()`, `sendfile()`). The server TCP stack sends as much data as the TCP congestion window (*cwnd*) allows in the context of the system call. The remainding of the output data will be sent as soon as *cwnd* widens, when TCP acknowledgements (ACKs) are received [13].

Thus, since some data is sent due to received ACKs, i.e., in the context of the thread[3] receiving packets, having multiple processors simultaneously sending data into the network requires having multiple processors simultaneously receiving packets from the network. Furthermore, the ratio

$$\frac{\textit{number of packets sent in the thread receiving ACKs}}{\textit{number packets sent in the transmit system calls}}$$

determines to what extent having multiple processors simultaneously receiving packets can lead to better performance. In effect, we see two factors that contribute to varying this ratio: network latency and zero-copy transfers. For a given connection, the higher the network latency between the server and its client, the more likely the output data will be in the TCP output queue by the time the ACKs arrive, so the lower the ratio. And also, with zero-copy transfers, i.e., without copying output data between user and kernel spaces (e.g. using `sendfile()`), the output data is guaranteed to be in the TCP output queue by the time ACKs arrive, so most segments will be sent out due to received ACKs.

In order to better understand where performance gains are to be expected we derive a simple model. Let T be the total processing time for a given HTTP request. Let T_e be the fraction of T spent due to network receive events (in the thread receiving packets), and T_a the fraction of T spent in the application (including both user and kernel modes). We neglect processing time spent in timers and therefore assume that $T = T_e + T_a$. Let p be the number of processors in the server machine.

If only one processor executes the network stack at a time, then the maximum achievable rate (in number of connections per second) equals:

$$R_1 = \begin{cases} \frac{1}{T_e} & \text{if } p > 1 + \frac{T_a}{T_e}, \\ \frac{p}{T_e + T_a} & \text{otherwise.} \end{cases} \quad (1)$$

Note that $R_1 = 1/T_e$ corresponds to the best case, when the number of processors is sufficient for the application to keep up with the network stack ($p > 1 + T_a/T_e$). If the number of processors is insufficient for the application to keep up ($p \leq 1 + T_a/T_e$) the maximum achievable rate is equal to $p/(T_e + T_a)$, which is lower that $1/T_e$.

Now, by assuming that both the network stack and the application scale linearly with the number of processors, if p processors can simultaneously process network receive events, the number of connections per second one can achieve is:

$$R_p = \frac{p}{T_e + T_a} \quad (2)$$

Therefore, using p processors results in a speedup equal to:

$$\alpha = \frac{p}{1 + \frac{T_a}{T_e}} \quad (3)$$

[2]The response is either directly retrieved from disk or memory, or dynamically generated. In the former case, the request is said *static*, it is said *dynamic* in the latter case.

[3]We are not referring to any operating system's execution context here, so the term "thread" must be understood in its broad sense.

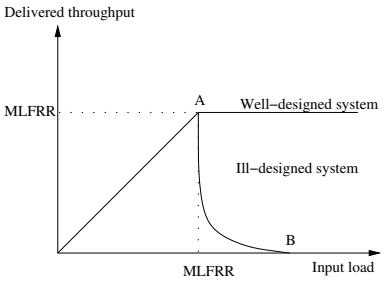

Figure 1. *Well-designed vs ill-designed systems.*

We note that the speedup α increases with the ratio T_e/T_a. This ratio corresponds to the percentage of CPU time spent in the kernel thread(s) processing incoming network packets over the percentage of CPU time spent in the application thread(s). It is akin to the ratio presented previously: the factors contributing to increasing/decreasing both ratios are similar. So, again, the higher the network latency, the more TCP segments are sent due to received ACKs, so the higher the ratio(s). And avoiding the memory copies between user and kernel spaces contributes to increasing the ratio(s) as well.

3 Robustness issues

Robustness problems can arise when a system is subjected to input overload. Ideally, the machine is sized in such a way it is able to handle the maximum input load the network on which it is attached can deliver. In real life, for cost considerations, server machines are sized to support a given mean input load. However, it is crucial that such machines behave gracefully under input loads above the mean load. This is especially true for Internet servers since Internet traffic is bursty in nature, with peak loads exceeding the average load by factors of 10 [14].

The throughput delivered by a server system must keep up with the input load until the server saturates. The server's peak throughput reached at the saturation point is called the *Maximum Loss Free Receive Rate* (MLFRR) [3]. Beyond the saturation point, the delivered throughput is expected not to drop below MLFRR. Figure 1 illustrates this. Point A corresponds to the saturation point. Beyond A, the well-designed server maintains constant throughput whereas the ill-designed one severely degrades. B corresponds to the point from which the ill-behaved server is no longer able to do any useful work, i.e., its delivered throughput is nil. Mogul et al offered first a complete study of this effect, they refer to it as *receive livelock* [3].

Kernels of operating systems execute in different contexts: *scheduler context*, *interrupt context*. In the rest of this section, we will refer to kernel control paths running in scheduler context and interrupt context as *scheduler threads* and *interrupt threads*, respectively. The kernel executes in scheduler context when it either executes on behalf of a user thread, as a result of a system call, or in a kernel thread[4]. The kernel executes in interrupt context when responding to a hardware interrupt. In Linux, scheduler threads (user threads either in user mode or kernel mode, and kernel threads) can be preempted by interrupt threads whereas interrupt threads cannot be preempted by scheduler threads.

NICs generate interrupts to notify the host operating system of incoming packets. Then, most operating systems, such as Linux and Solaris, process the incoming packets in interrupt context, thereby with the highest priority. The receive livelock becomes effective when the Receive Interrupts (RINTs) rate is so high that all CPU resources are spent handling RINTs and eventually dropping packets.

Mogul et al. first highlighted the receive livelock effect, and proposed a solution to eliminate it [3]. Their solution consists in combining the interrupt and polling modes. The NIC driver's Receive Interrupt Service Routine (RISR), which is responsible for taking care of interrupts caused by arriving packets or RINTs (Receive INTerrupts), disables the RINTs on the NIC so that subsequent arriving packets will not cause RINTs, and schedules the so-called *polling thread* for execution. When scheduled, the polling thread takes all packets present in the driver's receive queue through the network stack, each packet in turn. Once the polling thread has emptied the queue, it re-enables RINTs on the NIC. If the driver's receive queue fills up, i.e., the polling thread cannot keep up with the network, then the NIC drops packets, without consuming any CPU resources. We will describe these operations again shortly while presenting an implementation of this technique (NAPI).

Other more specialized solutions also exist: LRP[7] and SRP[8]. Basically, these techniques process incoming packets in low priority threads, namely in process or kernel-thread contexts. In addition to eliminating receive livelock, LRP and SRP aims to ensure fair allocation of system resources among the various applications that utilize the network. They achieve this by accounting and charging resources spent in protocol processing to the application on whose behalf this processing is performed.

Recently, Salim et al. implemented the solution proposed by Mogul et al. in Linux, they named their implementation NAPI (New API) [9]. NAPI adopts all the mechanisms proposed by Mogul et al.

Figure 2 shows the various components that come into play in NAPI. Upon receiving the packet, the NIC copies it into the receive queue in the driver's memory, and generates a RINT (Receive INTerrupt) if RINTs are enabled. The processor that takes the interrupt executes the driver's RISR

[4]Some kernels such as Linux and Solaris(TM) implement kernel threads. Kernel threads are scheduler's entities, they differ from user thread running in kernel mode in the sense that they do not run on behalf of a user thread, as a result of a system call.

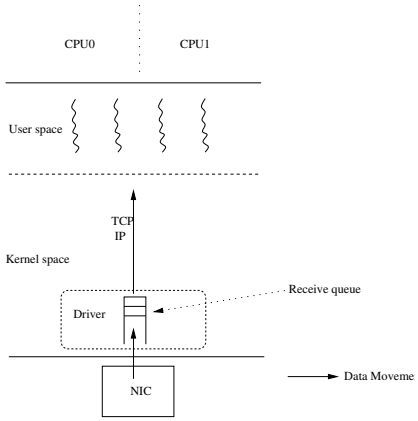

Figure 2. *NAPI components.*

(Receive Interrupt Service Routine)[5]. The RISR disables RINTs on the NIC so that subsequent incoming packets delivered by the NIC will not cause an RINT, indicates to the kernel that there is work for this device (by enqueueing the structure variable representing the device in the interrupted processor's device queue), and schedules a *softirq*[6] for further processing (TCP/IP processing). The softirq, which also runs on the interrupted processor, pulls the device off the device queue, and calls the `poll()` primitive on this device (`dev->poll()`). The `poll()` primitive, implemented by the NIC driver, is responsible for taking all packets present in the driver's receive queue through the network stack, each in turn. When the `poll()` primitive finds an empty entry in the queue, it re-enables RINTs in the NIC, removes the device from the device queue, and returns.

It is interesting to note that even though interrupts are distributed among the processors, no two packets are simultaneously processed with NAPI. Instead, a packets burst is processed by one processor, and once the entire burst has been processed, another burst may be processed by another processor[7]; but packet processing does not execute in parallel.

In addition to eliminating receive livelock, the solution proposed by Mogul et al. and implemented in Linux by Salim et al. provides other benefits. Under high input load, the polling thread happens to pull few packets off the driver's receive queue before re-enabling RINTs in the NIC, thus ensuring low latency. In contrast, under high input load, the polling thread can process lots of packets before re-enabling RINTs in the NIC, therefore increasing the number of packets per interrupt ratio and hence the throughput of the system. Also, not re-enabling RINTs during network processing allows to ensure fairness among the various NICs in the machine. Detailing all benefits of this solution is beyond the scope of this paper, interested readers are invited to refer to the appropriate documents [3, 9].

[5]Recent x86-based SMP machines include a so-called I/O APIC (I/O Advanced Programmable Interrupt Controller). By default, Linux programs the I/O APIC in such a way that each interrupt vector is distributed in a round-robin manner among the various processors.

[6]In Linux, packet processing occurs in *softirq* context, softirq corresponds to interrupt context with the serviced hardware device's interrupt vector re-enabled in the machine's interrupt controller.

[7]E.g. due to the interrupt round-robin algorithm in the I/O APIC.

4 New networking subsystem proposal

In this section, we first list the requirements of our networking subsystem, present our design choices to meet those requirements, and then describe our implementation, KNET.

4.1 Requirements

Our objective is to design and implement a parallel networking subsystem (a network subsystem capable of receiving and transmitting packets coming from and going to a single interface in parallel) that is efficient and robust. For our subsystem to be effective, we want to avoid cache and reordering issues that are of concern when processing network packets in parallel. And for it to be robust, we want to eliminate all possibilities for receive livelock (explained in section 3).

4.2 Design

Here we first explain the design choices to make our parallel networking subsystem efficient as well as robust, and then describe the overall functioning of our system by presenting the various steps for receiving packets.

4.2.1 Efficiency

Processing network packets in parallel raises instruction-cache and data-cache locality issues [15]. In particular, processing packets of the same connection on different processors results in cache misses when accessing connection-specific data (TCP Control Block). We maximize instruction- and data-cache locality by creating per-processor *network threads*, binding each network thread to a particular processor, and classifying incoming packets before they enter TCP so that it is always the same processor that processes all packets of a connection. In addition to maximizing cache locality, connection-level parallelism minimizes contentions on per-connection locks. Furthermore, processing all packets of a TCP connection on a single processor avoids reordering issues in TCP. Most TCP implementations use Jacobson's header prediction technique [16], which fails if packets arrive unordered.

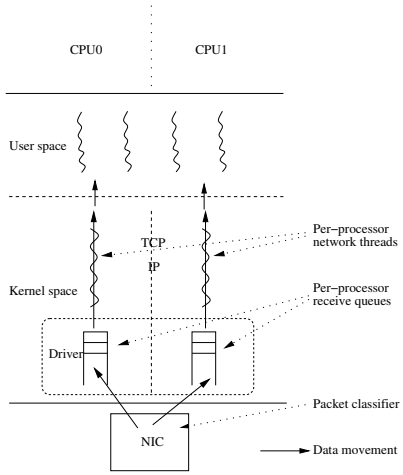

Figure 3. *KNET components.*

4.2.2 Robustness

Ensuring robustness, i.e., designing a livelock-free network subsystem, while performing connection-level parallelism as described above is challenging. In previous work on connection-level parallel network processing [4, 5, 6], incoming packets are classified in the kernel, more precisely in the RISR. As explained in section 3 re-enabling interrupt while processing packets can lead to receive livelock. We achieve robustness by implementing per-processor receive queues in the driver, having the NIC classify incoming packets and deposit the classified packets in the appropriate receive queue, and applying the technique proposed by Mogul et al. [3] that we have presented in section 3. Figure 3 shows the various components of our design.

4.2.3 Overall functioning

The NIC inputs a packet from the network, and classifies it to decide which driver's receive queue the packet should be copied into. After classification, the NIC copies the packet in the appropriate receive queue. At this point either the receive queue is currently being polled by its corresponding network thread or not. If it is currently polled then the NIC generates a RINT (Receive INTerrupt) and marks the queue *polled*. As long as the queue is marked *polled* the NIC will not generate RINTs as packets come in. If an RINT has been generated, the processor that takes it executes the driver's RISR (Receive Interrupt Service Routine), which retrieves the index of the receive queue, indicates there is work for this NIC, and wakes up the appropriate network thread. Once scheduled, the network thread retrieves a reference to the NIC, and takes all packets present in the corresponding NIC driver's receive queue through IP then TCP, each packet in turn. Once the receiving threads finds an empty entry in the polled receive queue it re-enables RINT in the NIC.

4.3 Implementation

To test the approach, we developed a software prototype on Linux and Myrinet. The prototype consists in modifications to the Myricom/Myrinet NIC firmware and driver and in an implementation of a Linux module.

Myrinet is a full-duplex $2 + 2$Gbps proprietary switched interconnect network commercialized by Myricom [17]. Figure 4 depicts a block diagram of a Myrinet NIC. Myrinet NICs are built around a RISC processor, namely the LANai. The firmware executed by the LANai is downloaded in the NIC memory at the time the NIC driver is inserted into the kernel. Myricom provides a software suite, GM, and a compilation suite, that, among other things, includes a cross-compiler for the LANai processor (`lanai-gcc`). GM provides drivers emulating Ethernet that can be used under the regular kernel TCP/IP stack. In our prototype, we modified GM-1.5's Linux "IP driver" and firmware.

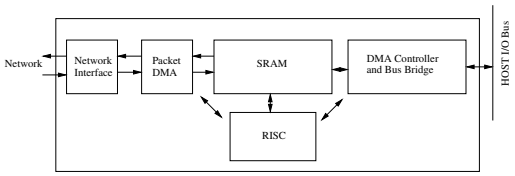

Figure 4. *Block-diagram of a Myrinet NIC.*

The modified NIC firmware classifies each incoming packet in order to determine which receive queue the packet should be DMAed into. In the current implementation, the classification function is $idx = ip\_src \,\&\, (n\_queues - 1)$, where ip_src is the source IP address of the packet, n_queues is the number of receive queues, and idx the resulting index of the appropriate receive queue. In addition to being trivial to implement, this packet classifier has the advantage of being stateless, yet resulting in good load-balancing among the processors given the large population of clients Internet servers must face under high load.

The modified NIC driver implements as many receive queues and receive buffer descriptors rings as processors. The new NIC firmware maintains copies of each receive buffer descriptors ring in its memory, and itself fetches the rings using DMAs. In the current implementation, the number of receive queues and rings is defined at compile time but we believe specifying it at open time should be feasible.

The networking subsystem itself is implemented in a Linux module[8]. At startup time, the Linux module creates as many kernel threads as processors (the network threads) and binds each to a particular processor. Each network

[8]Linux modules are objects that can dynamically be linked to a running Linux kernel.

thread then enters an event loop and goes to sleep waiting for network receive events from NICs. The driver's RISR is responsible for waking up the network threads. In contrast to NAPI which executes the TCP/IP stack in `softirq` context, KNET executes it in kernel-thread context. The reason for this is that we do not control onto which processors interrupts from the NIC arrive (because the I/O APIC is responsible for this). For example, it may occur that processor A takes the RINT whereas the packet (or packets burst) that caused the RINT is to be processed on processor B. In this situation, the RISR executing on processor A must wake up the receive thread on processor B. Achieving this would not be possible if TCP/IP were to execute in `softirq` context.

5 Experimental results

5.1 Experimental setup

In this section, we describe our software and hardware setup used for our performance measurements.

5.1.1 Hardware

Four 2-way PIII (600Mhz, 256KB L2 cache, 256MB SDRAM, ServerWorks CNB20LE Host Bridge) machines and one 4-way PIII (550Mhz, 512MB SDRAM, ServerWorks CNB20HE Host Bridge) are used throughout the experiments. The 4-way machine acts as the HTTP server and the four 2-way machines as the clients. The five machines are networked together through the Myrinet network.

5.1.2 Operating system configuration

All involved machines run a 2.4.20 Linux kernel. On the client machines, all OS-related settings are left to their defaults. On the server machine we change the *send socket buffer* to be 32 Kbytes in length. This value is chosen so that one call to `sendmsg()` or `sendfile()` can be sufficient to initiate the transfer of the response file. In all experiments we compare KNET to the NAPI version of the GM-1.5 Linux driver and firmware, which we implemented for the purpose of the comparison.

5.1.3 Benchmark programs

Two software programs are used for performance evaluation: Webfs [18] at the server side and a modified version of Sclient [19] at the clients side. We briefly describe both in the following.

Webfs is an event-driven HTTP server for purely static content, it uses the `select()` system call to wait for events without blocking. Though Webfs is not a purely multi-threaded server[9], it supports threads, where each thread runs its own `select()`/`accept()` loop. In addition, to maximize throughput and minimize CPU utilization, Webfs uses the *zero-copy* `sendfile()` system call.

The client machines run a modified version of Sclient. Sclient is an HTTP traffic generator specifically designed to generate HTTP requests rates beyond the capacity of the server, without employing a huge number of client processes. Using Sclient as a starting point, we implemented a new HTTP traffic generator which functions as follows. First, a user-defined number of TCP connections is generated towards the HTTP server. Then, for each open connection, an HTTP request is sent and the corresponding response is received. Once the response is received, a new TCP connection is initiated, etc. The `select()` system call is used to avoid blocking in the `sendmsg()` and `recvmsg()` system calls.

5.1.4 Workload

To minimize interactions with the server's file system each Sclient instance requests a different file. Since we use 4 Sclient instances in our experiments there are 4 different files served by the HTTP server, each being 20 or 5 Kbytes in size. Also, in all experiments, we vary the number of concurrent connections each client instance opens. We go up to 80 concurrent connections (4×20), which is sufficient to drive the server machine to saturation (0% idle CPU time) for every experiment case we ran.

5.2 Results and analysis

5.2.1 9000-byte MTUs

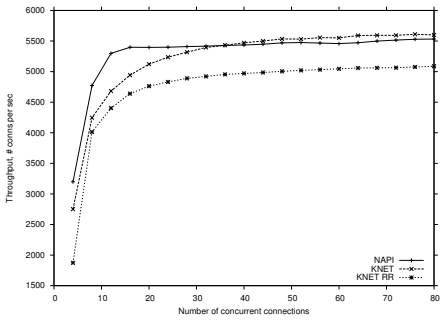

Figure 5. *Performance results of NAPI, KNET and KNET-RR, with* 9000*-byte MTUs,* 20*-Kbyte requested files, and* `sendfile()`.

Figure 5 reports the aggregated throughput delivered by the server versus the number of concurrent connections for 9000-byte MTUs. The `sendfile()` system call is used in all experiments here. First, we observe that KNET leads to about 3% improvement over NAPI. For 9000-byte MTUs,

[9] In the sense that Webfs does not use per-connection threads.

the application and network threads spend about 75% and 25% of CPU time, respectively[10]. For this sharing in CPU time and for 4 processors the model presented in section 2 gives a speedup $\alpha = p/(1 + T_a/T_e)$ equal to 1, the TCP/IP stack is therefore not the bottleneck for 9000-byte MTUs. The experimental results reported in the graph were therefore predictable. However, it is interesting to note that, even in cases where one processor suffices to process the network, KNET does not exhibit worse performance than NAPI. The curve KNET-RR represents the case where the NIC does not classify incoming packets. Instead, the NIC directs incoming packets to processors in a round-robin fashion. In the KNET-RR case, the processing of packets is still parallelized but, since packets of the same connection can be processed by two different processors, data-cache locality is not as good as with KNET. Peak throughput obtained with KNET-RR is effectively 20% lower than that obtained with KNET. This shows the benefits of classifying packets before they enter the network stack.

5.2.2 1500-byte MTUs

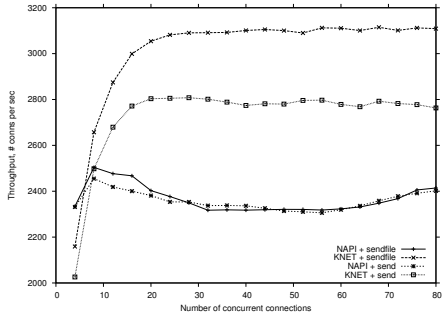

Figure 6. *Performance results of NAPI and KNET, for 1500-byte MTUs, 20-Kbyte requested files, with and without* `sendfile()`

Figure 6 reports the aggregated throughput delivered by the server versus the number of concurrent connections for 1500-byte MTUs. Here we report results of NAPI and KNET with and without `sendfile()`. KNET leads to 34% improvement over NAPI when `sendfile()` is used, and to 17% when using the regular `sendmsg()` system call. For 1500-byte MTUs, the application and network threads use roughly 55% and 45% of CPU time, respectively. With 4 processors, the model gives a speedup equal to 1.8. The experimental speedup ($\approx$ 1.3) is below the analytical one because we assumed in the model that the network stack scales linearly with the numbers of processors, which is untrue in practice. We will go back to this issue shortly. It is interesting to note that with NAPI using

[10]The CPU utilization numbers reported here obtained using the `top` command.

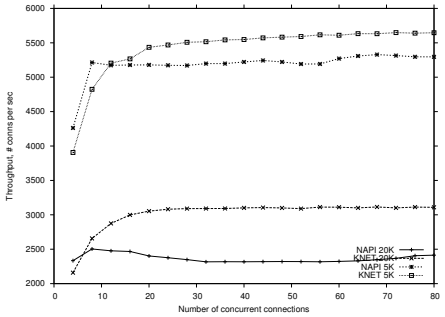

Figure 7. *Performance results of NAPI and KNET, with 1500-byte MTUs, and* `sendfile()`*, for 20-Kbyte and 5-Kbyte requested files.*

`sendfile()` or `sendmsg()` does not make any difference. Indeed, with NAPI the network stack executing only on one processor at a time is the bottleneck so minimizing the CPU time spent on behalf of the application threads does not result in better performance.

Figure 7 again reports the aggregated throughput delivered by the server versus the number of concurrent connections for 1500-byte MTUs, but with two different requested file sizes: 20-Kbytes and 5-Kbytes. `sendfile()` is used in all experiments here. For 5-Kbyte requested files, KNET leads to 6% improvement over NAPI, in contrast to the 34% improvement for 20-Kbyte requested files. This performance drop comes from the fact that the ratio T_e/T_a (corresponding to the percentage of CPU time spent in the kernel threads processing incoming network packets over that spent in the application threads) is lower for small files than for large files, because the smaller the response file, the fewer network packets.

Earlier, we stated that our model is not accurate because while building it we assumed that the network stack and application scale with the number of processors, which is untrue in practice. Figure 8 shows that the network stack does not scale because of the *listening* socket being accessed by all processors. Indeed, using 4 instances of the web server with KNET, each *listening* on a different port, results in 10% improvement over using one instance listening on a single port. A solution of the 1 port case would be to do the classification in such a way that all packets destined to the listening socket (SYN packets) are always processed by the same processor. The problem with this solution is that it may lead to a system that is not well-balanced. With NAPI, it does not make much difference whether having 4 different ports or a single one because the network stack is the bottleneck.

6 Conclusion and future work

In this paper, we claim that classifying incoming packets before they enter the operating system's kernel is key to im-

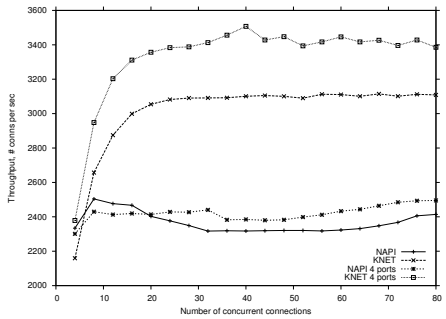

Figure 8. *Performance results of NAPI and KNET, with* 1500*-byte MTUs,* 20*-Kbyte requested files,* `sendfile()`*, with* 1 *and* 4 *webfs instances.*

plementing scalable and robust networking subsystem. We have designed and implemented a Linux networking subsystem built around a packet classifier in the Myrinet Network Interface Card. The experimental results reported in this paper show the relevance of our design and implementation when the network stack is the bottleneck, and that the performance of our implementation does not degrade when the network stack is not the bottleneck (e.g. for 9000-byte MTUs). In addition, we have derived a model allowing to predict if performance gains are to be expected when processing network packets in parallel. In particular, the model helped us understand our experimental results.

In our implementation, packet processing is achieved in kernel-thread context as opposed to interrupt context. We were constrained to resort to this solution because we want to be able to direct a packet to any processor regardless the processor that takes the interrupt. Processing the network in kernel-threads raises scheduling latency issues. Indeed, even if the kernel-thread has a high priority, it takes some time (the scheduling latency) for the thread to be scheduled by the scheduler. This scheduling latency can be observed at low load (4 concurrent connections) in the graphs reported in this paper. One solution to this problem is to design new hardware with which the classification-capable NIC can choose, based on the classification result, which processor to interrupt, as opposed to relying only on the machine's interrupt controller (I/O APIC in x86-based hardware).

As future work, we plan to further study the implication in terms of performance of operating the classification outside the kernel or not. In effect, we want develop to a software-based classifier and carry out extensive experiments. We also want to address the bottleneck issue due to the listening socket by proposing new classification algorithms or improving the operating system's kernel.

Acknowledgments

We would like to thank Erik Nordmark and Roland Westrelin from Sun Microsystems Laboratories for their invaluable help with this work.

References

[1] Ethernet Task Force. IEEE P802.3ae 10Gb/s, June 2002. `http://grouper.ieee.org/groups/802/3/ae/`.

[2] Intel Corporation. Intel PRO/10GbE LR Server Adapter, 2003. `http://support.intel.com/support/network/adapter/pro10gbe/pro10gbelr/index.htm`.

[3] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3), 1997.

[4] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994.

[5] J. D. Salehi, J. F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *IEEE/ACM Transactions on Networking*, 4(4), 1996.

[6] A. Garg. Parallel STREAMS: A multi-processor implementation. In *Winter 1990 USENIX Conference*, Washington, DC, January 1990.

[7] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation*, 1996.

[8] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. Signaled Receiver Processing. In *USENIX'2000 Annual Technical Conference*, San Diego, CA, June 2000.

[9] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *USENIX*, November 2001.

[10] H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. IETF, RFC1945, May 1996.

[11] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. IETF, RFC2616, June 1999.

[12] J. Postel. *Transmission Control Protocol*. IETF, RFC793, September 1981.

[13] V. Jacobson. Congestion avoidance and control. In *SIGCOMM'88*, Stanford, CA, August 1988. ACM.

[14] J. C. Mogul. Network behaviour of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[15] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache behaviour of network protocols. In *ACM SIGMETRICS'97 Conference*, June 1997.

[16] V. Jacobson. *4BSD Header Prediction*. ACM Computer Communication Review, April 1990.

[17] N.J. Boden, D. Cohen, and R.E. Felderman. Myrinet - A Gigabit-per-Second Local-Area Network. In *IEEE Micro*, volume 15 of *1*, February 1995.

[18] G. Knorr. Webfs. `http://bytesex.org/webfs.html`.

[19] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, 1997.