A New Mechanism for Transmission Notification on SMP

Eric Lemoine, Cong-Duc Pham, Laurent Lefèvre

To cite this version:

Eric Lemoine, Cong-Duc Pham, Laurent Lefèvre. A New Mechanism for Transmission Notification on SMP. [Research Report] RT-0295, INRIA. 2004, pp.16. inria-00069885

HAL Id: inria-00069885

https://hal.inria.fr/inria-00069885

Submitted on 19 May 2006

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A New Mechanism for Transmission Notification on SMP

E. Lemoine — C. Pham — L. Lefèvre

N° 0295

Mai 2004

Thème NUM

Rapport technique

ISSN 0249-0803

A New Mechanism for Transmission Notification on SMP

E. Lemoine[*][†] , C. Pham[†] , L. Lefèvre[†]

Thème NUM — Systèmes numériques
Projet INRIA/RESO

Rapport technique n° 0295 — Mai 2004 — 16 pages

Abstract: This paper presents the design, implementation, and evaluation of a new technique aimed to reduce the number of interrupts due to transmitted packets in the parallel network subsystem KNET. This technique combines interrupt and polling modes and relies on support in the Network Interface Controller — per-processor transmit queues — for distributing processing of transmission notifications among processors. Our prototype implementation on Linux and Myrinet yields performance gains of 16% on a 4-processor machine running a web server application.

Key-words: Parallel networking subsystem, Performance, Transmission notifications

[*] Sun Labs Europe
[†] INRIA/RESO

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Un nouveau mécanisme pour les notifications d'émission sur SMP

Résumé : Ce document présente l'architecture, la mise en œuvre et l'évaluation d'une nouvelle technique permettant de réduire le nombre d'interruptions dues à la transmission de paquets dans le sous-système réseau parallèle KNET. La technique combine les modes interruption et scrutation et repose sur de nouvelles fonctions dans le contrôleur réseau pour la distribution des notifications de transmission parmi les processeur. Notre prototype mis en œuvre dans Linux et Myrinet conduit à des gains en performance atteignant 16% sur une machine 4 processeurs exécutant un serveur Web.

Mots-clés : Sous-système réseau parallèle, Performance, Notifications d'émission

1 Introduction

With the continuing advances in network technology, much effort has been and still is devoted to optimizing network processing, in high-end systems. Network processing, traditionally performed by the operating system kernel, includes protocol (e.g. TCP/IP) processing, interrupt processing, buffer management, etc. Network processing optimization techniques rely on software optimizations only, while others require hardware support. For example, TOE and RDMA technologies [1], which are receiving lots of attention nowadays, require complex hardware assistance.

Our work concentrates on network processing on Symmetric Multi Processor (SMP). We use the term SMP here to refer to any architecture allowing multiple instruction flows to execute in parallel, including the emerging so-called multi-core, multi-threaded architectures. We think that these architectures raise new questions in networking and operating system design, and, therefore, conducting research in this area is essential. More specifically, the research work presented here seeks to develop and experiment with new techniques to optimize network processing on SMP.

We distinguish two approaches to network processing on SMP. In the first approach, a single processor[1] is responsible for network processing. This minimizes data movements between processor caches, and thus maximizes data cache locality. The second approach involves distributing network processing to multiple processors. This approach is more prone to cache misses, because more than one processor needs to access and write to the same memory locations. Nonetheless, we believe it is important to distribute network processing among processors for two reasons:

- Fairness. If one single processor deals with network processing, application threads executing on that particular processor have fewer CPU cycles at disposal than those executing on other processors.
- Scalability. With the continuing advances in network technology (e.g. [2]) and with the emergence of highly multi-threaded hardware architectures (e.g. [3, 4, 5]), one processor may not be able to keep up with network speed; applying multiple processors to network processing may therefore become a necessity.

[1] For the sake of clarity, we refer to processor, core, or hardware thread, as processor.

In previous work, we proposed a new network subsystem architecture that is built around a packet classifier in the Network Interface Controller [6]. The objective of this new architecture is to allow parallel packet processing, while maximizing cache locality and guaranteeing robustness. This is achieved by classifying incoming packets in the NIC, and assuring that all packets of TCP connection are processed by the same processor. In order to evaluate our architecture, we implemented a parallel network subsystem in the Linux kernel, named KNET, and a packet classifier in the programmable Myrinet NIC.

In this paper, we more specifically focus on reducing the number of interrupts due to transmitted packets in KNET. We suggest a technique that combines interrupt and polling modes, and, for efficiency, relies on support in the NIC to distribute the processing of transmission notifications among processors.

The rest of the paper is organized as follows. Section 2 describes the experimental setup used in our experiments. It also presents NAPI, the Linux standard network subsystem and KNET, our parallel network subsystem. Section 3 presents the traditional way of processing transmission notifications, and provides preliminary results in NAPI to highlight the performance gains that can be obtained using a technique combining interrupt and polling modes. Section 4 presents three strategies to reducing the number of interrupts due to transmitted packets in KNET, from the simplest to the most promising. Section 5 presents a performance evaluation of these three strategies. Section 6 highlights related work. Finally, Section 7 concludes and outlines future work.

2 Experimental environment

In this section, we present our experimental environment, and describe the two Linux network subsystems used in this work.

2.1 Hardware

The hardware platform used in all experiments reported in this paper is composed of:

- four 2-processor Pentium®III machines (600Mhz, 256KB L2 cache, 256MB SDRAM, ServerWorks CNB20LE Host Bridge),

File name	Size	Weight
file500.html	500B	350
file5k.html	5KB	500
file50k.html	50KB	140
file500k.html	500KB	9
file5m.html	5MB	1

Table 1: *WebStone file distribution.*

- one 4-processor Pentium®III machine (550Mhz, 1024KB L2 cache, 512MB SDRAM, ServerWorks CNB20HE Host Bridge).

The 5 machines are connected by a Myrinet network [7]. Myrinet is a full-duplex $2 + 2$Gbps proprietary switched interconnect network commercialized by Myricom. Myricom provides a Linux driver emulating Ethernet, which can be used with the regular TCP/IP stack, and a firmware program for the Myrinet NIC. Both are provided with source code, which enables to modify them at will.

2.2 Software

The 4-processor machine acts as an HTTP server, and the 2-processor machines as HTTP clients.

The client machines run Linux version 2.4.19, and execute WebStone2.5 [8] as HTTP traffic generator. WebStone generates HTTP/1.0 requests to the server using the file distribution of Table 1. With this file distribution, file500.html is requested 350 times out of 1000, file50k.html 140 times out of 1000, etc. In the experiments, we increase the load on the server by increasing the total number of concurrent connections opened by the clients. This workload allows us to saturate the server machine, we can therefore highlight performance implications in terms of delivered throughput.

The server machine runs Linux version 2.4.20, and executes the HTTP server Webfs [9]. Webfs is an event-driven HTTP server that uses `select()` for event handling, and `sendfile()` for zero-copy transfers. We use as many Webfs threads as processors in all experiments (4 threads).

2.3 Network subsystems

Two network subsystems are used on the server machine: NAPI [10] and KNET. NAPI (for New API) is the standard Linux 2.4.20 network subsystem. KNET is the parallel network subsystem for Linux that we developed. Below, we briefly describe the concepts and functioning of both.

NAPI aims at eliminating Receive Livelock [11], and more generally, reducing the rate of interrupts due to received packets. Receive Livelock occurs when the interrupt rate due to received packets is so high that the operating system does nothing but processing interrupts and eventually dropping packets, no other useful work can be performed. In the rest of this paper, we will refer to interrupts due to received packets as RINTs (for *Receive Interrupts*). NAPI works as follows. After receiving a RINT, the driver's *Interrupt Service Routine* (ISR) disables the RINTs on the NIC, and schedules a *network thread*. While the RINTs are disabled, the NIC continues to place incoming packets in the driver's receive queue (through DMA), but stops generating RINTs. Once scheduled, the network thread calls the driver's `poll()` method (`dev->poll()`). `poll()` is responsible for taking all packets present in the driver's receive queue through the network stack (e.g. TCP/IP stack). `poll()` re-enables RINTs on the NIC once it has emptied the receive queue. Note that, with NAPI, no two packets are simultaneously present in the network stack. NAPI exposes a new API to network drivers. To run experiments, we ported the Myrinet driver and firmware to NAPI.

KNET is our parallel network subsystem for Linux. KNET relies on packet classification under TCP to ensure that (almost)[2] all packets of a TCP connection are processed by the same processor, as well as on NAPI's principles to reduce the rate of RINTs and eliminate Receive Livelock. KNET uses per-processor *network threads*, each attached to a particular processor, and relies on per-processor receive queues in the driver. Every incoming packet is classified in the NIC and, based on the classification result, is placed (using DMA) in one of the receive queues. The classification ensures that all packets of a TCP connection are processed by the same processor. After receiving a RINT, the driver's ISR first reads from the NIC memory the index of the receive queue into which the packet causing the RINT has been placed. It than disables the RINTs on the NIC for that particular receive queue, and

[2] As opposed to packets sent due to arriving TCP acknowledgements, packets sent in the context of the application thread can be processed by any processor.

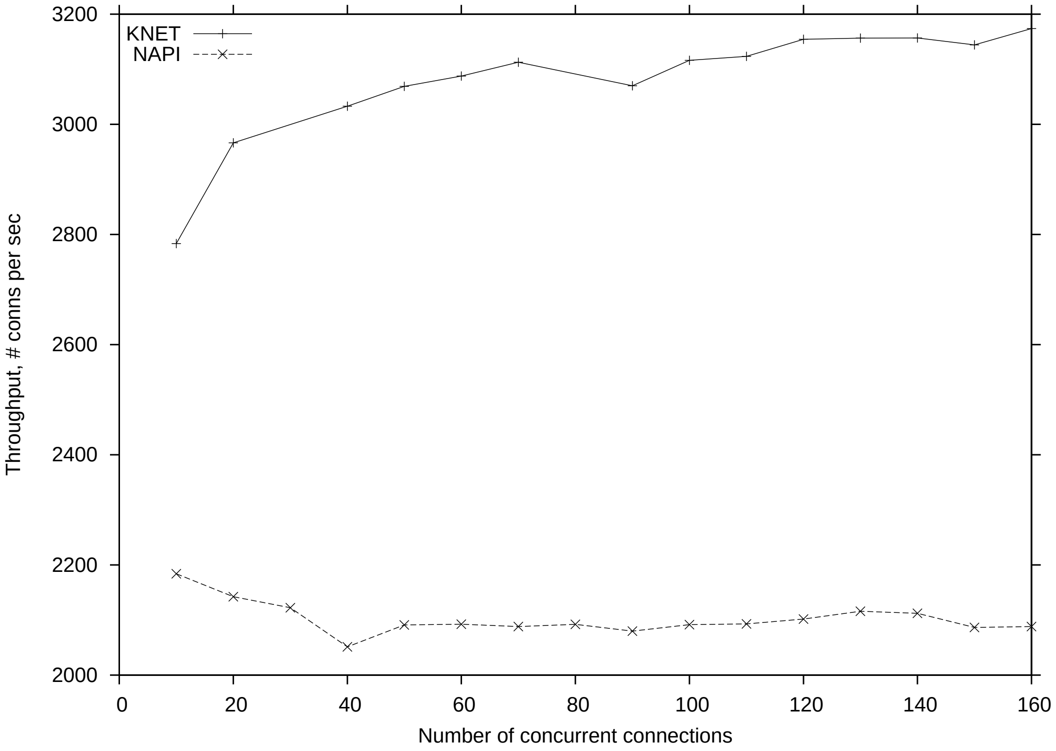

Figure 1: KNET *versus* NAPI.

schedules the network thread running the processor to which the receive queue is attached. Disabling the RINTs for a receive queue makes the NIC stop generating RINTs as it places incoming packets in this particular queue. Once scheduled, the network thread calls the driver's `poll()` method (`dev->poll()`). `poll()` takes all packets present in the driver's receive queue attached to the executing processor through the network stack. `poll()` re-enables the RINTs on the NIC for the processed receive queue once it has emptied it. Note that, as opposed to NAPI, multiple processors can simultaneously execute `poll()`, so more than one packet can be present in the TCP/IP stack at a certain time. As NAPI, KNET exposes a new API to network drivers. We implemented a KNET version of the Myrinet driver and firmware.

Figure 1 reports the throughput delivered by the HTTP server with NAPI and KNET. The delivered peak throughput is 45% higher with KNET than with NAPI. For a more detailed study of KNET, see [6].

3 Problem statement

Each time the NIC transmit a packet onto the network, it must somehow notify the host so that the resources associated with the transmitted packets can be released. In Linux, each network packet is represented by an object of type `struct sk_buff`, an `skb`, which must be de-allocated once the transmission of the associated packet is complete. Traditionally, NICs generate a *Transmit Interrupt* (TINT) every N packets, N being lower than or equal to the length of the *transmit queue*. Upon receiving a TINT, the driver's *Transmit Interrupt Service Routine* (TISR) deallocates the N `skbs` associated with the N transmitted packets. In this work, we attempt to eliminate this limit of N packets per TINT in our parallel network subsystem, KNET.

Upon receiving an interrupt, be it a RINT or a TINT, recent official Linux NAPI drivers' ISR disables both the RINTs and TINTs on the interrupting NIC before scheduling the polling thread. In addition to dealing with incoming packets, these drivers' `poll()` method walks through the transmit queue to check if transmission notifications have been DMA'd in by the NIC. Once `poll()` has processed all incoming packets and transmission notifications, it re-enables the RINTs and the TINTs on the NIC. To evaluate the performance benefits of polling transmission notifications, we modified our NAPI Myrinet driver along what is described above.

Figure 2 reports the throughput delivered by the HTTP server using NAPI with and without polling of transmission notifications. In each experiment—for each number of concurrent connections—the processor responsible for packet processing is 100% utilized. We observe that the peak throughput delivered by the server using polling is around 30% greater than without polling. Note that, with polling, one interrupt is generated at the beginning of each experiment and no subsequent interrupt is needed. In summary, this result suggests that implementing some similar mechanism in KNET may result in significant performance improvements.

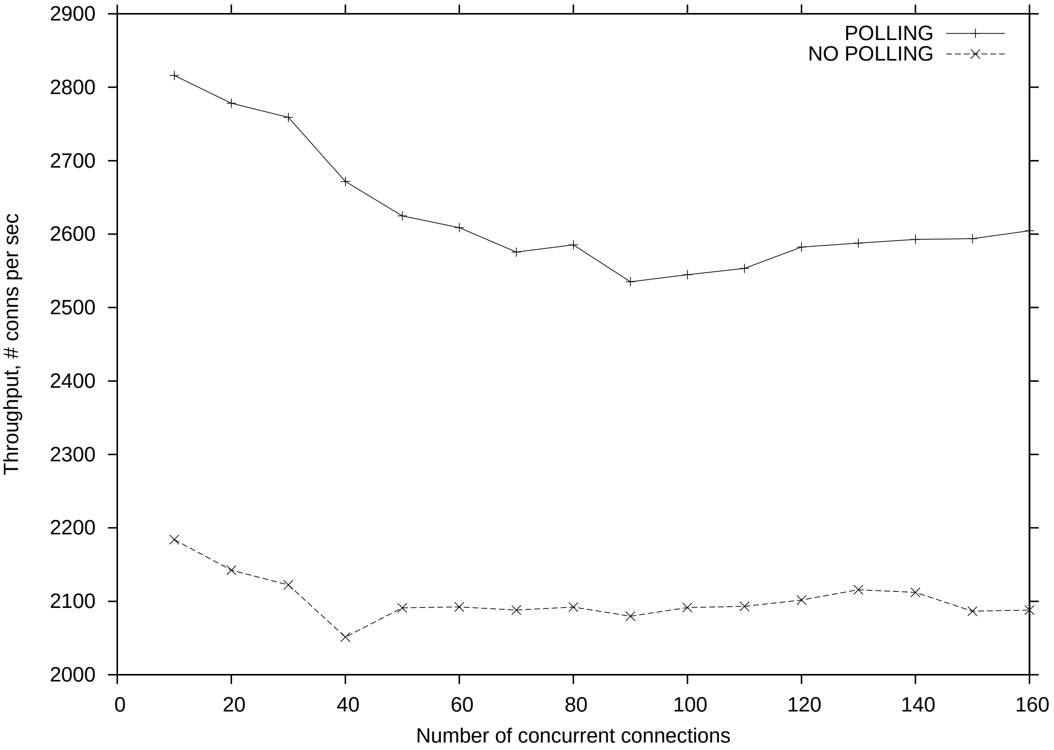

Figure 2: *Polling of transmission notifications in* NAPI.

4 Transmission notifications in KNET

In KNET, as opposed to NAPI, multiple processors can be executing `poll()` simultaneously. Thus, there are two possible strategies for polling of transmission notifications: either a single processor is dealing with transmission notifications at a time or multiple processors are. The first strategy is more straightforward as it is very close to the mechanism implemented in NAPI drivers. However, having only one processor dealing with transmission notifications has two problems. First, it can lead to an unbalanced network load distribution among processors, which may result in an unfair distribution of CPU resources between application threads. Second, it can result in a poor utilization of the memory allocator's Magazine Layer, as explained below.

In the TCP case, which is of particular interest here, `skbs` de-allocated after transmission of the corresponding packets are allocated in the TCP layer when TCP is about to send a packet. Linux, as other operating systems, uses the Slab Allocator [12] and the Magazine Layer [13], both initially implemented in the Solaris(TM) operating system. The Slab Allocator is an object cache that allows efficient object allocations and de-allocations. The Magazine Layer is a per-processor caching scheme aimed at reducing lock contentions on the Slab Allocator layer and keeping objects in processor caches between de-allocations and allocations. Given that `skbs` are allocated by all processors, using a single processor to de-allocate `skbs` (after transmission of the corresponding network packets) results in more de-allocations than allocations in this processor's Magazine Layer. Thus, the Magazine Layer inevitably fills up. The first de-allocation operation after the Magazine Layer is full is forced to empty it by putting the de-allocated objects in the Slab Allocator. In addition to increasing the number of executed instructions, this can create contentions on the lock serializing accesses to the Slab Allocator, and increase processor cache miss rates due to movements of objects between processors.

A more promising set of strategies consists of using multiple transmit queues, and distributing the processing of transmission notifications among processors. Here again, multiple approaches exist. One approach is to distribute transmit notifications in a round-robin manner. This allows to evenly distribute the load among processors, and ensures a better utilization of the Magazine Layer. Another, potentially better, approach is to distribute transmission notifications in such a way that every `skb` is de-allocated by the allocator processor, i.e, the processor that allocated it.

This tends to keep objects on the same processor and therefore reduce processor cache misses due movements of cache lines between processor caches. In addition, this allows for an even better distribution of allocations and deallocations in the Magazine Layer during periods when some processors transmit more packets than some others.

We implemented those three strategies in KNET. The first strategy was straightforward to implement in the sense that it is almost similar to the NAPI implementation. The last two strategies necessitated modifications to the NIC driver, as well as to the NIC firmware. Both basically involve using per-processor transmit queues in the driver, and passing the transmit queue index to the NIC each time a transmit request is passed to it. After transmission is complete, the NIC uses the transmit queue index to insert the transmission notifications in the correct transmit queue.

5 Performance evaluation

Figure 3 reports the throughput delivered by the HTTP server using KNET with and without using polling, and, when using polling, for the three strategies described above. Note that for each experiment—for each number of concurrent connections—all processors are 100% utilized.

We observe that the greatest peak throughput is achieved by POLLING-AFFINITY, which corresponds to the case where transmission notifications are distributed in such a way that every `skb` is de-allocated by the allocator processor. POLLING-AFFINITY leads to 16% improvement over the original KNET implementation (with no polling of transmission notifications). In between those are the strategies POLLING-ROUND-ROBIN, which corresponds to the case where transmission notifications are distributed among the processors in a round-robin manner, and POLLING-CPU0, where only CPU0 processes transmission notifications. POLLING-CPU0 yields a peak throughput around 5% below POLLING-AFFINITY and around 4% below POLLING-ROUND-ROBIN. We can notice that POLLING-AFFINITY only slightly improves performance over POLLING-ROUND-ROBIN. The effects of using affinity-based techniques here are limited by the small number of processors used. We believe that the gains obtained with POLLING-AFFINITY should increase with the number of processors.

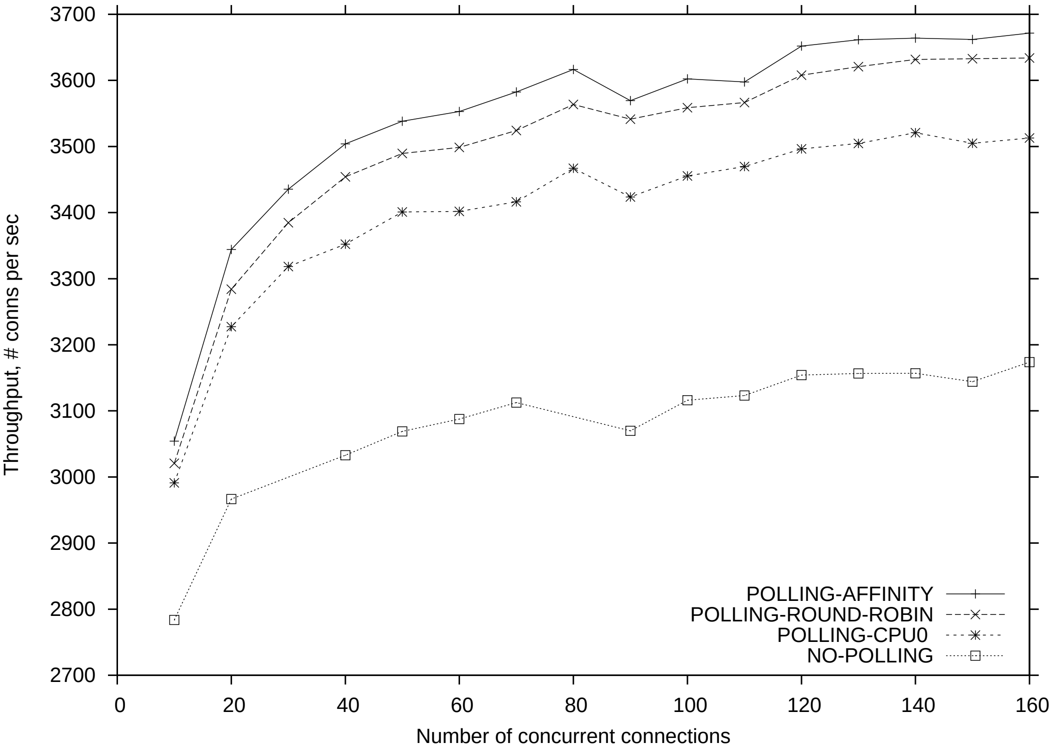

Figure 3: *Polling of transmission notifications in* KNET.

6 Related Work

Combining interrupt and polling modes to improve the efficiency and robustness of network subsystems was initially suggested by Mogul et al. [11]. Their work, which was used in the design of the, NAPI, the Linux network subsystem, by Salim et al. [10], focuses on the receive side. The work presented in this paper builds on this work and NAPI by applying it to the transmit side and to SMP architecture.

A lot of works on designing, implementing, and experimenting with high-speed NICs and high-performance host-to-NIC interfaces exist in the literature [14, 15, 16, 17, 18, 19]. These works focus on optimizing the partitioning of network functions between the NIC and the host system, and building efficient interfaces between hardware and software. Our work is similar to these works in the sense that we also suggest new functions into the NIC, but differs in the sense that we suggest NIC functions to improve network processing on SMP.

Salehi et al. studied the performance impact of various affinity-based scheduling approaches to parallel network processing [20]. In particular, they made use of *per-processor free-memory pools* to minimize cache misses on objects dynamically allocated and de-allocated during network processing. Our work is complementary to theirs in the sense that we suggest some NIC assistance to maximize the usage of the Magazine Layer, which is akin to their per-processor free-memory pools.

Nahum et al. studied the cache behavior of network protocols [21]. Their study indicates that caching is key to achieving high performance in network protocols, and that larger caches and increased cache associativity improve performance. However, they focused on uni-processor architecture. In contrast, we focus on improving cache locality on SMP.

7 Conclusion

In this paper, we have shown that substantial performance gains can be achieved by polling packet transmission notifications as opposed to using interrupts in the Linux standard network subsystem, NAPI, as well as in our parallel Linux network subsystem, KNET. We have demonstrated that distributing the processing of transmission notifications among processors in KNET results in larger performance gains than having a single processor dealing with the processing of transmission notifications. In addition, we have shown that distributing processing of transmission notifications in such a way that every object representing a network packet—every skb—is de-allocated by the allocator processor can lead to further gains. The technique involves passing the allocator processor's identifier to the NIC at transmission time and letting the NIC insert the transmission notifications directly in the allocator processor's transmit queue. We suggest that new NICs should support per-processor transmit queues so that the technique studied here can be applied.

In our prototype implementation, the processing of packets and transmission notifications is performed in kernel thread context as opposed to interrupt context. This constraint is due to hardware limitations preventing us from having the NIC interrupt any processor. One potential problem with processing NIC events in kernel thread context is the extra latency due to scheduling. Another problem is that locks must be taken in interrupt context to insert a work request and schedule the appropriate thread if it runs on a different processor than the interrupted processor. These two problems potentially reduce performance. With Message Signalled Interrupt (MSI) a device is capable of targeting an interrupt to any processor[3], so it would be interesting to carry out experiments with MSI-capable devices.

In this work, we have looked at improving cache locality on skbs allocated by the TCP stack at transmission time. As future work, we plan to explore the performance implications of ensuring that skbs allocated by the application threads also get de-allocated on the allocator processor. One possible solution would be to create some affinity between application and network threads by scheduling application threads on processors doing related network processing activities.

Also, we want to further investigate about the potential benefits and issues of multi-threaded architectures for network processing. In particular, we envision that making the NIC aware of the underlying architecture (number of processors, number of hardware threads) may make the NIC more capable of correctly placing events in memory. For example, the algorithms for distributing network processing to processors may differ depending on whether the processors share the cache or not.

[3] MSI and MSI-X are defined in the PCI-2 and PCI-X Specifications. Refer to http://www.pcisig.com for further details.

References

[1] J. C. Mogul. TCP Offload is a Dumb Idea Whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

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

[3] IBM. Power4 System Microarchitecture. `http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html`.

[4] Sun. Throughput Computing. `http://www.sun.com/processors/throughput/`.

[5] Intel. Hyper-Threading Technology. `http://www.intel.com/products/ht/hyperthreading_more.htm`.

[6] Éric Lemoine, CongDuc Pham, and Laurent Lefèvre. Packet classification in the NIC for improved SMP-based Internet servers. In *Proceedings of IEEE 3rd International Conference on Networking (ICN'04)*, Guadeloupe, French Caribbean, March 2003.

[7] 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.

[8] Mindcraft. Webstone2.5. `http://www.mindcraft.com/webstone/`.

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

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

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

[12] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, 1994.

[13] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference*, 2001.

[14] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *SIGCOMM*, 1994.

[15] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: A network-independent card provides architectural support for high-performance protocols. *IEEE Network*, 7(4), July 1993.

[16] K. K. Ramakrishnan. Performance Considerations in Designing Network Interfaces. *Selected Areas in Communications, IEEE*, 11(2), 1993.

[17] B. S. Davie. The Architecture and Implementation of a High-Speed Host Interface. *Selected Areas in Communications, IEEE*, 4(4), 1993.

[18] R. M. Metcalfe. Computer/Network Interface Design: Lessons from Arpanet and Ethernet. *Selected Areas in Communications, IEEE*, 4(4), 1993.

[19] G. W. Neufeld, M. R. Ito, M. Goldberg, M. J. McCutcheon, and S. Ritchie. Parallel Host Interface for an ATM Network. *IEEE Network*, 7(4), 1993.

[20] 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.

[21] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache Behavior of Network Protocols. In *ACM SIGMETRICS '97 Conference*, June 1997.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803