Studying Network Protocol Offload With Emulation: Approach And Preliminary Results

Roland Westrelin*, Nicolas Fugier*, Erik Nordmark*, Kai Kunze* and Eric Lemoine*
*Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
Email: *Firstname.Lastname*@Sun.COM

***Abstract*— To fully take advantage of high-speed networks while freeing CPU cycles for application processing, the industry is proposing new techniques relying on an extended role of the network interface card such as TCP Offload Engine and Remote Direct Memory Access. This paper presents an experimental study aimed at collecting the performance data needed to assess these techniques. This work is based on the emulation of an advanced network interface card plugged on the I/O bus. In the experimental setting, a processor of a partitioned SMP machine is dedicated to network processing. Achieving a faithful emulation of a network interface card is one of the main concerns and it is guiding the design of the Offload Engine software. This setting has the advantage of being flexible so that many different offload scenarios can be evaluated. Preliminary throughput results of an emulated TCP Offload Engine demonstrate a large benefit. The emulated TCP Offload Engine indeed yields 600 to 900% improvement while still relying on memory copies at the kernel boundary.**

I. Introduction

Due to the continual advances in network technology, many studies aimed at optimizing network processing in end-systems have been carried out over the last 15 years. Examples are [1] and [2].

Today, Network Interface (NI) vendors are beginning to commercialize so-called TCP Offload Engines (TOEs) which are NIs capable of executing TCP/IP. The vendors claim that this capability will be necessary to handle data flow at link speed with emerging networking technology (10 Gigabit Ethernet). If it turns out that the overhead of processing the non-data touching in TCP/IP represents a high percentage of processing time, as indicated in [3], TOE could be an interesting solution for relieving the host processor workload. There is, however, certain controversy over the benefits of TOE. TOE is believed to bring its own performance and deployment issues [4]. Also, a number of studies have shown that data-touching operations within the end-system, including in particular memory copies are the major source of overhead [5], [6]. In addition, although a TOE can facilitate the elimination of memory copies for applications directly built on top of TCP, it is of limited help in eliminating memory copies if an Upper Layer Protocol (ULP) with its own headers and data payload is used.

Remote Direct Memory Access (RDMA) over IP [7] is a new protocol suite that provides a generic "built-in" solution to achieving zero-copy [8]. The RDMA protocol consists of a Direct Data Placement (DDP) component which performs the actual transfer of the packet into the correct buffer memory. RDMA implies extended NI processing, a new protocol stack and comes with new APIs. RDMA/IP is a technology strongly related to TOE because, in a typical implementation of the RDMA protocol stack, the added layers need to be in the NI to achieve early demultiplexing of incoming messages. Thus, the TCP/IP protocol layered below also needs to be in the NI.

Full protocol offload is not new [9], [10], [11]. However, it has never succeeded for complex general-purpose protocols such as TCP/IP. The current renewed interest in TCP offload is explained by a change in the performance trade-offs. Because interaction between the host and the NI can be performed in units independent of the packet size with a TOE, a large decrease in per-packet overhead should be observed for long transfers. This gain could potentially increase with network speed because the maximal packet size does not increase as fast as link speed (still 1500 bytes for 10 Gigabit Ethernet). It could be argued that by the time faster networks are widely deployed, fast enough processors will be available [12]. However, note that it is not sufficient that the system can sustain the full network throughput. Some cycles must be left for application processing. Furthermore, if the current trend towards multi-threaded multi-core CPUs as the new path to increased processor performance is confirmed then the fast processors of the future might not be of much help for single connection performance. Indeed, TCP processing has been shown to be efficiently parallelizable only at the connection level [13]. So in the future, systems might be able to fill a big pipe with many simultaneous connections but not with a single one.

From a more pragmatic angle, NI-level TCP implementations will need to become a commodity for RDMA or iSCSI solutions to become widely accepted. Once an offload engine is available, the extra effort required to provide a performance boost for any applications by making it a generic TOE is tempting.

Our research has indicated that performance data around TOE and RDMA/IP is lacking. Therefore, this study aims to experimentally study TOE, RDMA/IP and possible alternate solutions embedded in a network interface card on the I/O bus, so that educated choices can be made. This paper presents an original methodology to attack the problem. Preliminary

0-7803-8686-8/04/$20.00 ©2004 IEEE

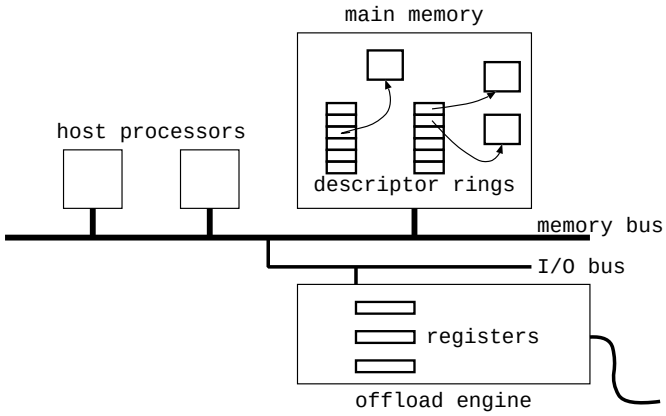

Fig. 1. The targeted architecture: an Offbad Engine on the I/O bus which uses typical mechanisms to interface with the host.

results are also given: although more complex protocols such as RDMA are in the scope of this study, only TCP offload numbers are presented here.

The rest of the paper contains the following sections. Section II introduces the goals of this study. Section III presents the experimental setting used in evaluating results. In Section IV, preliminary results are presented. Section V contains the conclusions of this preliminary study.

II. Project Context

The experimental study presented in this paper, named the TOAD project, aims to evaluate the performance issues in the TOE and RDMA/IP design space. The targeted technologies include full TCP Offload, extension to a full TOE for one specific ULP, RDMA implemented in a full TOE, NI-level acceleration without a full protocol offload in the case of a specific ULP or RDMA. NFS is one major target application both for the ULP-aware designs and as an RDMA application [14].

In the scenarios of interest to TOAD, the Offload Engine is always a network interface card plugged on the I/O bus as illustrated by Figure 1. It follows the typical choices of such designs. The network interface is programmed through registers located on the card and accessed over the I/O bus (Programmed I/O). It uses DMA to read and write main memory and is able to send interrupts to host processors. Descriptors for buffers to be used for incoming or outgoing messages are stored in rings located in main memory.

Questions that the project is addressing:

- What performance is gained from offloading the TCP/IP protocol to the network interface? From removing memory copies?
- What are the performance issues in designing the host/offload engine interface? Careful design of the interface between the host and the offload engine is key to performance.
- What performance advantages would an Upper Layer Protocol (ULP) aware TOE bring? How does an ULP aware TOE compare with RDMA solutions?
- How do the extra operations brought by direct data placement techniques (memory registration, extra message round trips) affect performance?
- Is putting TCP/IP in the NI the only way of designing high-performance RDMA-enabled or ULP-aware NIs? A different approach could be to leave TCP/IP processing in the host operating system and to design the NI in such a way that it enables direct data placement by parsing incoming packets.

Even though this study could produce conclusions valid for any platform, experiments carried out in the TOAD project focus on Sun SPARC®/Solaris™ machines. Account is taken for the specificities of these machines such as the existence of IOMMUs[1].

III. Experimental setting

Relying on existing TOE products to explore the TOE/RDMA space has constraints: only a few products are available, accessing the product's internals, if possible, is inflexible, and the limits of the product can restrict the validity of the results. Thus, the TOAD project takes a different approach and relies on emulation for the performance study.

A. Principle of the emulation

The key idea is to partition a SPARC/Solaris SMP machine and to use some of the processors as an offload engine (OE). Real packets are still exchanged with other machines over a dedicated network interface.

Relying on emulation has many advantages. Because the OE is a piece of software, it is infinitely customizable: any offload scenarios and the design of the best host/OE interface (a key problem in this design space) can be explored . Understanding the performance results is also made easier because probes can be inserted in any part of the system.

This idea is not really new since it is used, for instance, in the ETA prototype [15]. The TOAD approach is unique in that this project builds an experimental setting that is as close as possible to the real hardware situation, and in understanding and quantifying where reality and theory differ. It is not a project aimed to investigate a new way to use the resources of an SMP machine but to emulate the behavior of an OE that would be plugged on the I/O bus.

In this article, the processors running the OE code are referred to as the *OE processors*. The remaining processors that run application and Solaris code are called the *host processors*. Communication between machines occurs through a *real NI* by opposition with the *emulated NI* composed of the real NI and the OE processors.

The constraints in achieving a faithful emulation are:

1) If m processors of a n-processor machine are used for the OE emulation, the "virtual" $(n-m)$-processor machine must behave as close as possible to a true

[1]The IO Memory Management Unit is a piece of hardware enabling DMA on virtual memory addresses

$(n-m)$-processor machine. This is an issue because the m OE processors and the $(n-m)$ host processors share resources.

2) The CPU cycles used by the host processors when interacting with the OE must be as close as possible to those of interacting with a real OE.

One simple way of running OE code on the machine would be to call the Solaris TCP code from a Solaris kernel thread bound to one processor. This violates Constraint 1 because the execution of the OE would disturb the remaining processors. At the operating system level, the OE code would share code and many data structures with the Solaris operation system and the host processors would experience contention on them. The only way to fully avoid this problem is for the OE not to run any Solaris code. Also because the OE processors need to be fully dedicated to their task, they cannot receive and process interrupts. Finally, managing the n processors of the machine has a cost: cross-calls (inter processor function calls built on inter processor interrupts) are used to keep the state of all the processors consistent. Because they are synchronous and because all processors managed by the system are often targeted, they have a cost for the initiating processor which depends on the number of processors managed by the system. To comply with Constraint 1, cross calls should not be sent to the OE processors.

Even if the OE code somehow runs outside of the Solaris operating system on its own processors, the host and OE processors still share the memory bus in the same coherency domain. So the host processors can be slowed down because of memory traffic caused by the execution of the OE (the OE accessing its code or data). Isolating the OE processors from a hardware perspective means minimizing cache and memory bus interactions. Most UltraSPARC® III/IV machines feature the Sun™ Fireplane Interconnect [16] as a memory bus. Processors and memory are organized by processor board: each CPU module comes with memory, access is interleaved across the memory that is on the same processor board. The data path is switched but addresses use a shared bus. For instance, a small 4-processor machine (the one used for the experiments) has two processor boards with two CPUs and memory. By dedicating one full processor board (memory and CPUs) for the network processing, memory traffic is kept local to the board. Only coherency traffic is then visible in the whole machine, but there is nothing to be done about it (see Section IV for an evaluation of the coherency traffic effect).

B. Implementation

The OE code is built in such a way that it does not need any support from the Solaris operating system and can run stand-alone. It is based on a BSD TCP stack (FreeBSD 4.8): this implementation has proven to be simple, robust and fast. It is organized around an event loop: the OE processor keeps on polling for events coming from the network, from the host processors or from an internal timer. The software is not multi-threaded and only uses one CPU. However, all the CPUs of the OE processor board are put off-line for an optimal isolation from the rest of the machine. The OE code features its own generic memory allocator (based on Solaris slab/vmem allocator [17]). Time is updated by reading the tick register of the processor.

The machine used for the test boots a modified version of the Solaris operating system. Early during the boot process, the memory of one processor board is stolen from the Solaris operating system. The machine comes up with all the processors running the Solaris operating system. To start the OE:

1) The memory of the processor board that is to be used by the OE is mapped using the largest possible pages at a fixed offset in the address space of the kernel, referred to as the OE memory.
2) The code of the OE (a stand-alone binary file) is loaded in its memory.
3) Configuration data is passed to the OE through a predetermined location in OE memory.
4) A Solaris kernel thread is created.
5) The processors on the processor board chosen to run the OE are put off-line. This uses a standard Solaris feature: a processor can be designated to run no threads and to receive no interrupts. It still participates in cross-calls that are necessary to keep the processor in a consistent state with respect to the rest of the system. The processor is then waiting in a tight loop to be put back online.
6) The processors of 5 are then removed from cross-call participation: they won't be in a consistent state and putting them back online is not safe.
7) The kernel thread of 4 is forced to run on one of the off-line processor (the OE processor) and starts executing the OE code.

At least, one extra network interface is installed in the machine. The Solaris driver for this NI discovers this interface and performs the basic configuration. A driver in the OE (largely similar to a FreeBSD driver) takes the control of this NI when the OE is started.

On the sending side, the OE programs its real NI so that data is sent directly from the host buffers in host memory to the network. Checksum offload has to be supported by the real NI for this to make sense. On the receiving side, packets are received to OE buffers allocated from OE memory so that the placement of incoming data is under OE control. Data is then copied to host memory.

Since the experimental platform fully controls whether memory gets allocated from host memory or OE memory, it reproduces an host/OE interface that very closely matches that in the real hardware: descriptor rings in host memory, "registers" in OE memory. Transfers performed by the OE processor (emulated DMA) to read descriptors from host memory, write completion events to host memory or write data to host buffers are performed using block loads and stores [18]: data movements that do not allocate in the cache and thus do not introduce false sharing of cache lines.

Interrupts from the OE to the host processors are emulated using inter-processor interrupt: a mechanism that is identical to device interrupt in SPARC/Solaris.

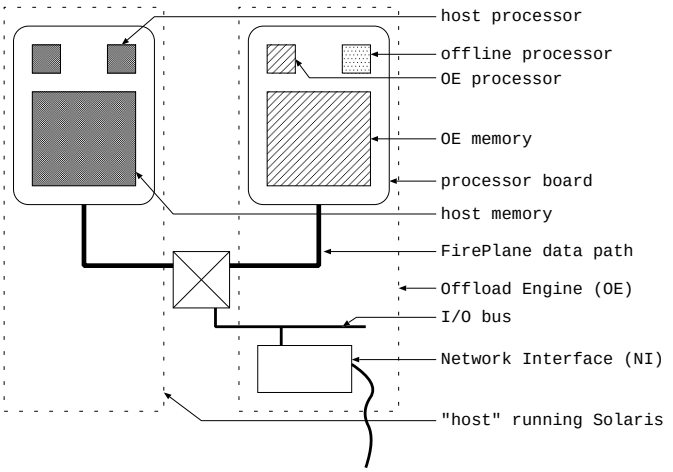

Fig. 2. The architecture used in the emulation of an Offbad Engine: a full processor board (2 processors and memory) and a Network Interface are hijacked from Solaris and run a stand-alone event-driven piece of software emulating an OE.

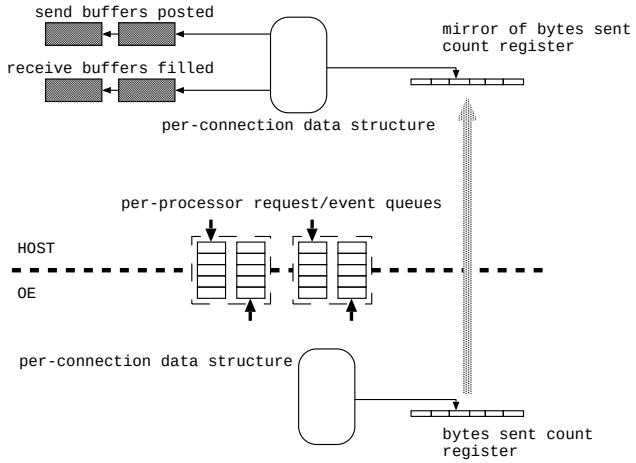

Fig. 3. Elements of the interface between the host and Offbad Engine.

When a memory mapping is destroyed, cross-calls are used to invalidate the TLBs (Translation Look-aside Buffers) of all processors. Because the OE processor is not kept consistent with the rest of the machine running the Solaris operating system through cross-calls, it needs to access host memory using persistent virtual mappings. Thus, all of the memory used by the host processors is mapped once and for all before the OE is started and the OE only uses this stable mapping to access host memory. Because UltraSPARC processors are 64-bit, a large part of the address space is still available after Solaris has set up its own mappings and creating those large mappings is not an issue.

Figure 2 summarizes the architecture of the experimental platform.

C. Host/Offload Engine interface

A poorly designed interface between the host and the OE is likely to ruin most of the performance benefit brought by protocol offload. This section presents the choices that were made to design an efficient interface. The description is mostly limited to the elements needed to understand the performance discussed in section IV. They correspond to the operation of the OE driver and OE to move large chunks of data in and out of the machine. The interface with the application is the BSD socket API: the application calls in the operating system, data is copied at the kernel boundary, hardware mechanisms provided by the OE are hidden from the application. The interface design would be largely different to enable full user-level communications.

The design described here was implemented in software in our emulated platform. Because the emulation is faithful and provides emulated variants of regular hardware mechanisms (Direct Memory Access, Programmed I/O), the interface could be implemented similarly in hardware and would bring the same performance benefits. In the following, those mechanisms are referred to as DMA and PIO even though in our implementation we used the emulated variants.

Figure 3 summarizes the interface.

The interface was designed to relieve as much as possible the main processors even if that meant putting an extra burden on the OE.

Most of the exchanges between the host and OE go through a request/event queue pair. The host posts request to the request queue and they are read by the OE. Similarly the OE posts events to the event queue and they are read by the host. Both queues are in host memory. They are accessed by the OE using DMA. The host notifies the OE of a new request in the queue by writing (PIO) to an OE register. One of the field of each event is used as a flag to identify that a new event is ready: by checking this flag in the current event queue head, the host knows whether a new event is ready or not. The OE can also trigger an interrupt to signifies the host that a new event was posted.

One such a queue pair exists per-processor: this way one processor can access its own queue pair with no contentions on lock.

The queue pair is used for every socket operations that are synchronous (when the host needs an answer from the OE to complete the operation): opening a socket, connect, listen, accept or closing a socket for instance. The host passes a request down to OE using the request queue and gets a reply back through the event queue. The calling thread is put to sleep and woken up by an interrupt triggered by the OE once the event is posted. As shown in Figure 3, the host and OE keeps their own per-connection data structures. The requests and events carry enough information for them to relate a request or an event to a particular connection.

Sending and receiving works differently.

To send, the host passes a descriptor of a kernel buffer to the OE through the request queue. But it does not wait for an event back, the thread does not block, it returns right away as expected by a BSD socket application. Notification

of completion of sends must be provided by the OE so that the host knows when to free buffers. This is achieved in a cumulative fashion: the OE keeps the count of the number of bytes sent for a particular connection. This count is mirrored in host memory using DMA. Freeing of buffers is performed lazily by the host: it reads the count of bytes sent and frees the buffers that it can. No interrupt is triggered. For the host, obtaining the information on the completion of sends in a cumulative fashion is much cheaper than if it had to process events from the OE: it does not have to go over many events to release several buffers in one pass.

BSD send blocks once a thread has filled its "socket buffer". That happens when the network consumes data more slowly than the application produces them. In our design, similarly, a thread cannot fill kernel buffers indefinitely if they are not processed fast enough by the OE. When the host notices that a thread has exhausted its share of buffers for a particular connection, this thread is put to sleep and the host request an event and an interrupt from the OE when enough data for this connection has been sent and acknowledged. This request, passed through the event queue, is independent of the size of the buffers passed to the OE (for instance, the host requests an event once 128KB of buffers is freed). When the host gets the event back, it wakes the application thread up.

Receiving is performed through fixed-length anonymous buffers. A large pool of buffers is provided by the host to the OE. They can be used for any connection. When data for a connection is received by the OE, it picks one of the anonymous buffer up, fills it and passes it back to the host through an event and an interrupt. In its interrupt routine the host demultiplexes incoming buffers: it attaches the newly filled buffer to the corresponding connection data structure. The application, in kernel space, will look at this connection data structure for new data. For the tests presented in this paper, the OE actually waits until it receives enough data to fully fill an anonymous buffer until it notifies the host that data has arrived. Of course, the host needs to periodically refill the anonymous buffer pool.

IV. Preliminary results

The test machine is a Sun Fire™ v480: two processor boards, each with two 1050MHz UltraSPARC III Cu processors (four processors total) and 4GB of main memory (8GB total). The network card is plugged on a 64bits/66MHz PCI bus. Note that the PCI bus implementation in this machine does not reach the theoretical maximal throughput of 4.2Gb/s. Peak throughputs of 1.12Gb/s from host to network and 3Gb/s from network to host are reported by a test provided by Myricom[2].

Drivers for two network interfaces were developed:

Myricom's Myrinet adapter. Myrinet [19] is a 2Gb/s full duplex proprietary interconnect technology popular in high performance computing. The Myrinet NI appears as a regular Ethernet NI from the IP stack perspective. In the driver, Ethernet packets are built and embedded in Myrinet frames that provide extra routing information (Myrinet uses source routing). Even though Myrinet implements back-pressure flow control at the link level, packet losses under congestion are still possible on the links or a the endpoints.

Sun Gigaswift Ethernet adapter, Sun's standard gigabit Ethernet NI.

The Solaris version used is a development version of the next official release (Solaris 10).

As mentioned in Section III, memory bus coherency traffic created by the OE software could interfere with the host processors and impact benchmark results thus breaking faithful emulation of a smart NI on an I/O bus. The results of a small test program in which one thread per host processor copies a large piece of data several times from and to the host memory are used to obtain an estimate of the worst-case perturbation. The goal is to measure the performance of this benchmark both when the OE processor is idle and when it is also copying blocks of data from and to its own memory. Note that because copies are performed inside host memory or inside OE memory, the data traffic created by the host processors and the OE processor cannot interfere (the data path on the memory interconnect is switched). Since the blocks copied are much larger than the caches, the test creates a lot of coherency traffic. The results show that the host processors are slowed down by less than 1% when the OE is loading the memory bus (from a mean memory copy bandwidth for the host copy of 642MB/s to 637MB/s). Furthermore, the test conditions are certainly much more demanding of the memory bus than any of the benchmarks presented here.

The network performance results provided here are obtained with the Iperf throughput benchmark [20]. It is a socket API test. A single connection is opened for the duration of the test: only the maximal throughput on a long-lived connection is benchmarked. The results include system calls and one copy on the sending and receiving sides through kernel buffers.

In the tests, Myrinet uses 9KB packets and Sun Gigaswift Ethernet adapter uses regular 1.5KB Ethernet frames.

What is interesting is not so much the maximal throughput as quantifying how using a TOE relieves the host CPUs. To be sure that comparisons are fair between the different networking technologies and TCP implementations, the load is expressed as the percentage of one of the machine's host CPU that is needed to achieve 1Gb/s of throughput. The cycles saved from using a TOE allow the system to run application code. Also, assuming that the I/O subsystem and networking hardware could be scaled up, the load of the host processors also gives an indication of the maximal throughput that could be sustained by one processor: if n% of one processor is necessary to handle 1Gb/s, $(100/n)$Gb/s could be sustained by the processor.

The baseline for comparison is obtained with the Solaris TCP stack and the Sun Gigaswift Ethernet adapter NI. The results are: 945Mb/s for 77% of one CPU on the sending side (81% of one CPU per Gb/s) and, 942Mb/s for 136% of one

[2] `http://www.myri.com/fom-serve/cache/121.html`

CPU (i.e. one CPU fully loaded and another 36% loaded) on the receiving side (144% of one CPU per Gb/s). The host TCP stack saturates the link in both directions. By tuning the configuration options of the Solaris kernel the throughput to CPU utilization ratio on the host CPUs can be maximized. When the application and the interrupts are all handled by the same processor, on the receiving side, only 741Mb/s of throughput is sustained for 99% of one CPU. That corresponds to 134% of one CPU per Gb/s. Even though the link is not saturated the CPU is more efficiently used. Table I and the following use this number (134% of one CPU per Gb/s) as the baseline number of the Solaris TCP stack.

With Myrinet, when sending through the TOAD OE, the throughput is 1.1Gb/s. As mentioned above, the PCI bus implementation does not offer theoretical peak throughput. It is the limiting factor here on the sending side. When receiving through the OE, the throughput is 1.8Gb/s. This result on the receiving side is on par with the best numbers published by Myricom[3]. With the Sun Gigaswift Ethernet adapter and the OE, the maximal throughput is 890Mb/s on the sending side and 940Mb/s on the receiving side.

However, the results obtained in the OE case with the Myrinet adapter and the Sun Gigaswift Ethernet adapter are the same when normalized to the throughput. This is as expected: the interactions between the host and OE determines the load of the host CPU and they are independent of the underlying network technology. When passing data in 128KB chunks between the host and OE, 11% of one CPU per Gb/s on the sending side is necessary, 12% of one CPU per Gb/s on the receiving side. Assuming the I/O subsystem and networking hardware can handle it, the host processor could handle close to 10Gb/s when 100% loaded. This represents an improvement factor of 7 on the sending side and an improvement factor of 10 on the receiving side as compared to Solaris TCP. When passing data in 8KB chunks, the load increases to 14% of one CPU per Gb/s on the sending side and 24% of one CPU per Gb/s on the receiving side. Passing data in larger chunks between the host and OE helps decrease the load of the host CPUs. Note that the load increases faster with smaller chunks on the receive side than on the send side. This is because, as explained in Section III-C, on the receive side the host is interrupted once per chunk when on the send side, cumulative notification is used and interrupts are triggered only when the application thread is put to sleep.

Table II summarizes the results.

In addition, note that: on the sending side, the user-to-kernel memory copy accounts for about 10% of one CPU per Gb/s, whether TOE is used or not does not affect the results. On the receiving side, the kernel-to-user memory copy accounts for about 10% of one CPU per Gb/s when using the TOE and about 22% when using the standard Solaris TCP stack. The higher overhead for the copy in the latter case comes from the extra overheard incurred when copying small data chunks (1460B chunks here).

[3] `http://www.myri.com/myrinet/performance/ip.html`

	Sending side	Receiving side
Memory copies	9%	22%
TCP/IP + driver	72%	112%
Total	81%	134%

TABLE I

SOLARIS TCP – LOAD ON THE HOST PROCESSORS EXPRESSED AS THE PERCENTAGE OF ONE CPU TO REACH 1GB/S

	Sending side		Receiving side	
	128KB chunks	8K chunks	128K chunks	8K chunks
Memory copies	9%	10%	10%	10%
TOE driver	2%	4%	2%	14%
Total	11%	14%	12%	24%

TABLE II

TOE – LOAD ON THE HOST PROCESSORS EXPRESSED AS THE PERCENTAGE OF ONE CPU TO REACH 1GB/S

V. CONCLUSION

This paper presents the use of emulation by partitioning an SMP machine to evaluate different protocol offloading scenarios. The results so far, restricted to a TOE used on a throughput test, show surprisingly large benefits for this very controversial technology. A single copy through kernel buffers is still included in these tests. Contrary to conventional wisdom, in this study, the copy overhead is not the dominating contribution to overheads.

However, the benchmark used in this paper was expected to behave favorably with a TOE. Future work includes a study of

other workloads : short-lived connections, transactional or latency-sensitive tests.

other offbad scenarios : processing specific to one ULP in the TOE, a zero-copy interface to the OE, RDMA.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their thoughtful comments on this paper. Many thanks also to James P.G. Sterbenz for his advices.

REFERENCES

[1] H. K. J. Chu, "Zero-copy TCP in Solaris," in *USENIX 1996 Annual Tech. Conf.*, Jan. 1996.

[2] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End System Optimizations for High-Speed TCP," *IEEE Communications*, vol. 39, no. 4, Apr. 2001.

[3] J. Kay and J. Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," in *SIGCOMM*, 1993.

[4] J. C. Mogul, "TCP offbad is a dumb idea whose time has come," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003, http://www.usenix.org/events/hotos03/tech/full_papers/mogul/mogul_html/index.html.

[5] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, June 1989.

[6] D. Clark and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the SIGCOMM'90 Symposium on Communications Architectures and Protocols*, Sept. 1990.

[7] IETF RDDP working group documents. [Online]. Available: http://www.ietf.org/ids.by.wg/rddp.html

[8] J. P. G. Sterbenz and G. M. Parulkar, "Axon: A Distributed Communication Architecture for High-Speed Networking," in *Proceedings of IEEE INFOCOM'90*, San Francisco, CA, June 1990, pp. 415–425.

[9] H. Kanadia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," in *ACM symposium on Communications architectures and protocols*, 1988, pp. 175–187.

[10] G. Chesson, "XTP/PE design considerations," in *IFIP Protocols for High-Speed Networks*, H. Rudin and R. Williamson, Eds. Elsevier/North-Holland, 1989, pp. 27–33.

[11] M. Zitterbart, "A multiprocessor architecture for high speed network interconnections," in *INFOCOM'89. Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, Apr. 1989, pp. 212–218.

[12] C. Partridge, "How Slow is One Gigabit per Second?" *ACM Computer Communication Review*, vol. 20, no. 1, pp. 44–53, Jan. 1990.

[13] 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, Nov. 1994.

[14] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad, "NFS over RDMA," in *Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, Karlsruhe, Germany, Aug. 2003, http://www.acm.org/sigcomm/sigcomm2003/workshop/niceli/papers/nfsrdma-paper.pdf.

[15] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong, "ETA: Experience with an intel xeon processor as a packet processing engine," in *Hot Interconnects 11*, Stanford University, Aug. 2003, http://www.hoti.org/Hoti11_program/papers/hoti11_11_regnier_g.pdf.

[16] A. E. Charlesworth, "The sun fireplane system interconnect," in *SuperComputing SC2001*, Denver, CO, Nov. 2001, http://www.sc2001.org/papers/pap.pap150.pdf.

[17] J. Bonwick and J. Adams, "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources," in *USENIX Annual Technical Conference*, 2001, pp. 15–33.

[18] *UltraSparc III Cu user's manual*, Sun Microsystems, Jan. 2004, http://www.sun.com/processors/manuals/USIIIv2.pdf.

[19] N. Boden, D. Cohen, and R. Felderman, "Myrinet - A Gigabit-per-Second Local-Area Network," in *IEEE Micro*, ser. 1, vol. 15, Feb. 1995, pp. 29–36.

[20] "Iperf - the tcp/udp bandwidth measurement tool," http://dast.nlanr.net/Projects/Iperf/.