IsoStack – Highly Efficient Network Processing on Dedicated Cores

Leah Shalev, Julian Satran, Eran Borovik, Muli Ben-Yehuda
leah@il.ibm.com, Julian_Satran@il.ibm.com, borove@il.ibm.com, muli@il.ibm.com

IBM Research – Haifa

Abstract

Sharing data between the processors becomes increasingly expensive as the number of cores in a system grows. In particular, the network processing overhead on larger systems can reach tens of thousands of CPU cycles per TCP packet, for just hundreds of "useful" instructions. Most of these cycles are spent waiting – when the CPU is stalled while accessing "bouncing" cache lines of network control data shared by all processors in the system – and synchronizing access to this shared state. In many cases, the resulting excessive CPU utilization limits the overall system performance. We describe an IsoStack architecture which eliminates the unnecessary sharing of network control state at all stack layers, from the low-level device access, through the transport protocol, to the socket interface layer. The IsoStack "offloads" network stack processing to a dedicated processor core; multiple applications running on the rest of the cores invoke the IsoStack services in parallel, using a thin access layer that emulates the standard sockets API, without introducing new dependencies between the processors. We present a prototype implementation of this architecture, and provide detailed performance analysis. We demonstrate the ability to scale up the number of application threads and scale down the size of messages. In particular, we show an order of magnitude performance improvement for short messages, reaching the 10Gb/s line speed at 40% CPU utilization even for 64 byte messages, while the unmodified system is choked when driving 11 times less throughput.

1. Introduction

While networking demands in data centers continue to grow, and the networking infrastructure continues to provide improved bandwidth and latency, single processor performance remains the same and in some cases even decreases. Recently, increasing the number of CPU cores became the only way to perform more instructions per cycle. However, the overhead due to interaction between these cores also goes up, and naïve data-sharing may inhibit performance scaling as the number of cores grows. Nevertheless, the familiar shared memory programming model is still commonly used for both application programming and implementation of OS services.

Since the days of uniprocessor systems, network processing has been carried out in a "multithreaded" fashion: some portions of the stack are executed during the socket system calls (in the context of calling applications), others during receive packet processing (in the context of interrupt handlers or kernel threads owned by the network stack), and yet others in the context of timeout handler routines. As multiprocessors were introduced, it was natural to distribute these stack processing elements symmetrically on the multiple processors in order to keep pace with the growing networking speeds. As the number of processors grows, the cost of sharing the network control structures between the processors becomes extremely high; meanwhile, cores become so abundant that sparing a few becomes feasible. This has provided an opportunity to re-think the network stack architecture and take advantage of the changing landscape of computer systems.

The IsoStack is a different approach for integrating network processing within a multicore system. Instead of using the cores symmetrically, the IsoStack uses dedicated cores for network processing, and leaves the rest of the cores for running applications. Since the network processing is confined to dedicated processors, the stack can be optimized – executed serially without interrupts and locks. Since the CPUs are not shared between applications and the stack, there are fewer context switches, and the cache behavior is improved. The IsoStack provides applications with a high-level interface (similar to a TCP Offload Engine interface), which can also allow efficient virtualization support using simple HW devices.

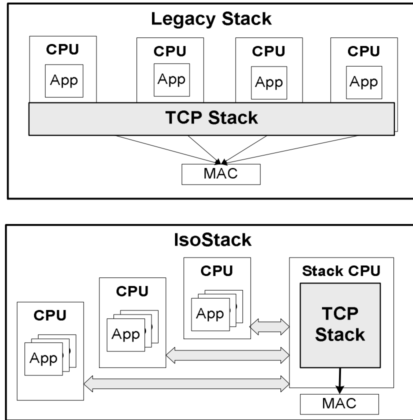

Figure 1. Native stack vs. IsoStack

The contributions of this paper are:

- The architecture of an isolated network stack that allows independent, contention-free, execution of TCP/IP control operations on a dedicated core, and application data processing on the other cores;
- The prototype implementation of such a stack in AIX 6.1 on Power6, providing a standard synchronous socket API built upon an asynchronous internal interconnect;
- Implementation of an optimized message queue mechanism for internal communication between a large number of applications (producers) and a consumer running on a dedicated core;
- The performance evaluation for a 10 Gb/s link, demonstrating a significant increase of bandwidth and/or decrease of total CPU utilization compared to the native stack, in some cases yielding an order-of-magnitude improvement.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 describes the system architecture, and Section 4 depicts the prototype implementation. We present the experimental results in Section 5, and conclude the paper in Section 6.

2. Background and Related Work

For decades, TCP performance optimizations were introduced gradually to address the performance hot spots of contemporary systems ([1, 2, 5]). The most widely adopted optimizations include checksum calculation offload, interrupt mitigation to decrease the number of interrupt requests from networking devices, and techniques that decrease the number of packets to be processed for bulk data transfer. Some of these techniques for decreasing the number of packets include jumbo frames ([5]), Large Send Offload (LSO [31]), also called TCP segmentation offload (TSO), and, recently, Large Receive Offload (LRO [21, 25]). Nevertheless, the resulting improvements merely succeeded to compensate for the rapidly growing networking demands, combined with relatively slow growth of CPU speed and even slower improvement of memory bandwidth and latency ([6, 16]).

With the advent of multiprocessor and (later) multicore systems, stack parallelization became necessary to keep pace with the growing network bandwidth. However, efficient parallelization remains challenging, as the parallel stack architectures implemented in the modern operating systems incur additional locking overhead, cache inefficiencies, and scheduling overhead ([23]).

Receive-Side Scaling ([18]) and similar techniques let a NIC classify the incoming packets to determine the affinity between these packets and CPU cores. On the basis of the packet classification result, received packets are dispatched to the appropriate receive queue, which is usually served by a particular processor. This technique allows more efficient low-level device sharing, as it relieves the bottleneck associated with sharing a single receive queue, and instead allows the stack to process received packets in truly parallel way when the packets are independent (i.e., belong to different sets of network flows). On special-purpose systems (such as embedded network appliances), running customized applications, this could potentially allow to confine all TCP processing for a particular connection to a single processor core. However, on general-purpose systems (running regular sockets applications), if the rest of the sharing issues are not addressed, RSS (as well as other receive-side optimizations such as NAPI) only allow to eliminate a small part of the multiprocessing overhead. This is because the receive processing, the transmit processing and the timer processing for the same TCP connection are still likely to be executed on different processors. In particular, application-triggered data transmission is executed in application thread context, while ACK handling and ACK-triggered data transmission are executed by the receive handler. The transmit thread either does not have any CPU affinity, or its affinity is configured by the application, while the affinity of the receive handler is configured by the operating system, transparently to the application. Also, an application thread can handle multiple connections, that can be mapped by RSS to different CPUs. Accordingly, such

un-coordinated execution still necessitates locking to protect access to the TCP connection and the associated socket state, and may cause cache line bouncing when accessing this state.

A radical solution to the fast-network, slow-host phenomenon is offered by RDMA approach ([10]). It offloads the protocol to an RDMA-enabled adapter, which allows zero-copy operation due to RDMA semantics, and eliminates per-packet overhead due to offloaded transport processing. Although this approach is suitable for high-performance computing applications running in a closed environment and using MPI or explicit RDMA semantics API, it is not feasible for data-center applications using sockets API, implementing standard protocols (such as HTTP) directly over TCP, and interacting with legacy clients. For this latter class of applications, pure TCP offload (without RDMA semantics) has been proposed.

TCP offload for socket applications has been pursued for a long time ([8, 11, 12, 13, 19]), and remains controversial. Its potential advantage is the improved performance due to a higher-level interface that decreases the amount of interaction between the software and the TCP Offload Engine (TOE) adapter, since the internal events are handled by the TOE adapter and do not disrupt application execution. However, in practice, the performance potential of TOE materializes only under various limitations. For example, it may be necessary to modify the existing applications in order to achieve improved performance. Also, due to high complexity and low volumes, TOE solutions tend to have high cost and longer development cycle comparing to the rest of the system components, which can make a TOE engine obsolete by the time it is released. In addition, TOE solutions lack the flexibility in protocol processing that is needed to support future protocol changes, and are prone to bugs that cannot be easily fixed. Even if the internal implementation is programmable, the changes can only be done by the adapter vendor, leaving the OS very little control over the protocol behavior. This impedes TOE support in some operating systems, and hinders TOE acceptance in general.

“TCP onload” using a dedicated CPU was proposed for multiprocessor systems as an alternative to TCP offload, without the disadvantages of hardware-based TCP offload ([14, 15, 17, 20]). The concept is based on an asymmetric multi-processing mode, where at least one of the CPUs on a multiprocessor system is dedicated to network stack processing, serving as an integrated TCP offload engine. This architecture allows a significant reduction in overhead when compared to naïve parallelization approaches. The TCP Servers project ([7]) also demonstrates the value of a similar approach. However, the previous solutions for CPU-based TCP offload made simplifying assumptions on the interaction between the applications and the onloaded stack, and did not demonstrate performance improvement for inconveniently small message sizes or for high number of applications sharing the "onloaded" services. The IsoStack work is focused on improving these aspects of the onload concept.

Loosely coupled TCP acceleration ([22]) is a hybrid approach that combines the benefits of both offload and onload. Similar to the offload approach, the application CPU uses a lightweight interface to interact with an “offloaded” network stack. However, network stack processing is not fully offloaded to the network interface adapter. Instead, only the data processing is performed by a hardware acceleration engine on the adapter, while the protocol control operations are done by software on a dedicated main CPU. The software and hardware components are loosely coupled; the parallelization is done in a way that allows asynchronous and independent operation of both parts. In particular, the control information that has to be accessed by both entities is replicated rather than shared, using message queues to explicitly exchange state changes.

The same principle of dividing up responsibilities was also applied in the Scalable I/O project ([26]), which showed that efficient and scalable I/O virtualization becomes possible by complete separation of the I/O and compute functions. Moreover, the OS structure itself can be revisited to reduce unnecessary sharing, as in the Corey operating system for many cores ([27]); or to eliminate the sharing altogether, as in the Multikernel architecture ([28]). Asymmetrical OS structure was also employed in the Piglet operating system ([4]) which used dedicated processors to implement "intelligent device" functions.

3. IsoStack Architecture

In this section we present the IsoStack architecture, in which we confine the network protocol processing to dedicated processors and isolate it from the application execution environment.

The IsoStack architecture is guided by the following design principles:

- Serialized, event-driven, lock-free, and interrupt-free implementation of the IsoStack on one or more dedicated logical processors. In particular, adapter control structures are not shared between processors.
- Asynchronous interaction between applications and the IsoStack, through explicit messaging, without the sharing of state.
- The isolation is transparent to applications; in particular, the underlying asynchronous protocol does not affect the latency of synchronous operations.

The first two design principles allow more efficient implementation of the network stack, with better utilization of multiple processors. This is due to elimination of the overhead caused by access to shared data structures from different processors and better use of each processor's resources (e.g., decreased cache pollution). The last principle allows unmodified applications to benefit from the improved stack performance, without having to switch to a different API or make any other changes.

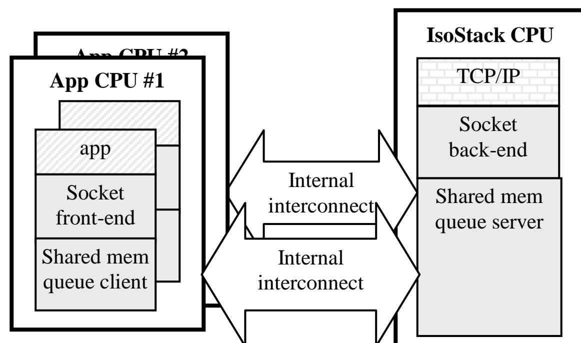

Figure 2. IsoStack architecture

The IsoStack architecture is depicted in Figure 2. Applications access network services using a socket front-end layer that implements the standard socket API and replaces the legacy sockets layer. The socket front-end handles the API peculiarities and delegates the execution of networking operations to the socket back-end. The socket front-end and the socket back-end interact using an asynchronous protocol over an internal interconnect. The architecture allows different types of internal interconnects. In our earlier work ([26]), we used Infiniband ([9]) for communication between the socket front-end and back-end. This work focuses on a message queue mechanism using the available general-purpose hardware; namely, cache-coherent memory; the detailed discussion is in Section 4.1.

Socket back-end receives network commands from socket front-end, executes the commands asynchronously and sends the command status in the opposite direction. The commands include socket transmit/receive/control commands, and buffer registration commands. Different APIs, such as standard synchronous BSD sockets or various flavors of asynchronous sockets, can be implemented using the same underlying command/status mechanism. For example, the asynchronous Extended Sockets API ([33]), which exposes explicit memory registration of application buffers, allows transmit implementation with true zero-copy. The standard socket API can be implemented with a single data copy into the socket transmit buffer, using in-advance registration of that internal socket buffer, as described in Section 4.2.

The IsoStack uses a dedicated logical CPU, and is solely responsible for all network processing for a particular network interface, which eliminates contention on access to network control data structures and allows a wide range of optimizations. Since the processor is not shared with other components, context switching overhead is reduced, and polling-mode interrupt-free execution becomes possible, eliminating the interrupt handler overhead. Since the data structures are not shared with other processors, single-threaded, serialized execution enables lock-free operation, thus eliminating the locking overhead. Consequently, all major sources of stack inefficiency are removed.

Although this paper focuses on the case of a single IsoStack processor and a single network interface assigned to it, this is not an architectural limitation. It is possible to run multiple independent IsoStack instances, where each IsoStack instance is responsible for one or more network interfaces. Moreover, since hardware support for packet classification (with multiple receive queues) is common, throughput scaling for a single network interface can be achieved by using several independent instances of the IsoStack, each responsible for a subset of network traffic flows on that interface, as discussed in Section 6.

On the other hand, it is not necessary to consume completely a processor core under light load. In order to save power when the traffic rate is low, the IsoStack can temporarily enable the interrupts and stop the polling until it is notified on a new event. The interrupt handlers in this case are used only to resume the polling, hence this type of interrupt-driven execution does not re-introduce the shortcomings of the regular stack implementation.

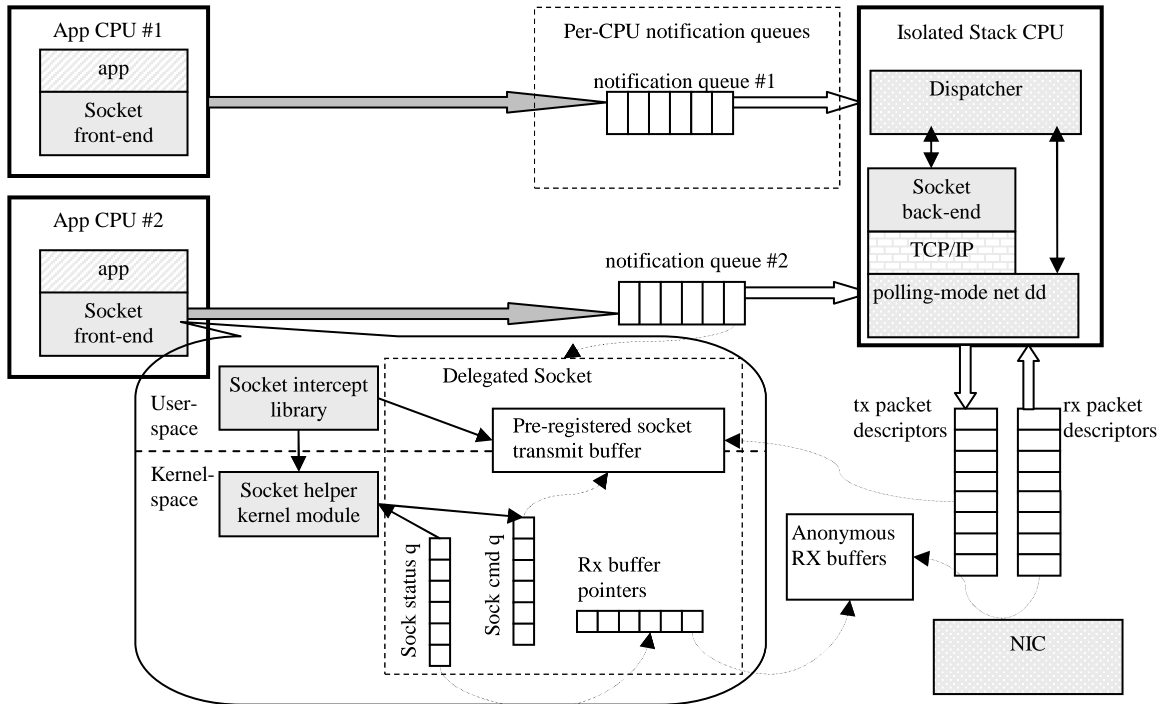

Figure 3. System Design

4. Prototype Implementation

The IsoStack prototype is based on the AIX 6.1 operating system, running on a Power6 system using the HEA 10Gb/s adapter. We modified several kernel components to allow the isolated-mode operation of the network stack as a single kernel thread, added new kernel extension modules to support "delegation" of socket operations to the IsoStack, and implemented a user-space library that intercepts socket operations and passes them to the IsoStack instead of invoking the socket system calls. Figure 3 depicts the high-level system design.

The socket layer is split into socket front-end and socket back-end to accomplish the delegation of socket operations. In particular, the state of each socket is split into its socket delegation state at the front-end, while the actual socket object (including the network protocol control information) is maintained at the socket back-end. The socket front-end consists of a socket intercept library that primarily provides user-space implementation of standard socket calls, and a socket helper kernel module that facilitates communication between the socket front-end and back-end when kernel-level privileges are required (for example, to access shared notification queues, as explained in Section 4.1). The socket back-end is a part of the IsoStack; it receives socket commands from the socket front-end, and executes them using the asynchronous in-kernel socket APIs adapted for single-thread, interrupt-free operation.

Section 4.1 describes the design of the messaging mechanism used for the interaction between the socket front-end and back-end. Sections 4.2 and 4.3 provide details of the transmit and receive operations, respectively. Section 4.4 describes the event-driven operation of the IsoStack. Section 4.5 lists the lock elimination optimizations enabled by our architecture.

4.1 Message Queues

An efficient mechanism for interaction between the application and the IsoStack is critical for realizing the performance improvement potential of our architecture. Clearly, executing the network processing on a separate CPU, without the overhead of locks or interrupts, reduces the stack overhead. However, the separation introduces a new overhead, which must be kept very low in order to make the overall solution worthwhile. In particular, this necessitates a highly efficient many-to-one producer-consumer mechanism, to pass commands to the IsoStack from multiple applications.

The design of such a mechanism was one of the main challenges of this work. Our early experiments showed that the existing IPC services are too expensive in terms

of both CPU utilization and latency. On the other hand, the existing solutions for lock-free, producer-consumer interaction via shared memory provide much better performance for low numbers of producers, but do not scale well as the number of producers grows, because the consumer must poll large numbers of queues. Ideally, a simple hardware mechanism could be employed to safely serialize request submissions from multiple non-cooperative, non-trusted clients to a single request queue, which could then be polled by the server. Such a mechanism could allow lock-free direct access to the queue by multiple producers, with atomicity handled by the hardware. Unfortunately, such a mechanism is not yet available, which makes the single queue approach unfeasible. The access to such a single queue becomes very expensive under the heavy contention due to the queue sharing by all socket applications (and all processor cores) in the system.

To decrease the cost of queue sharing, we chose to use a separate queue per logical processor (processor core or thread if SMT is in use). Thus, the number of queues is constant and small enough to allow efficient polling by the consumer. Each thread accesses (atomically) the queue of the processor on which it is running at the time of the access; the queue is not shared by other processors in the system, which allows contention-free producer operation. Unfortunately, since these queues are shared by different applications, they cannot be accessed directly from user-space; kernel-space socket helper provides protected access to the notification queues.

The per-CPU queues are used to notify the IsoStack of new application requests; the notification queue entries include only the socket identification information. The actual socket commands are kept in per-socket command queues that reside in shared memory, accessible to both socket front-end and socket back-end; the command responses are returned through per-socket status queues. The queues are implemented using the coherent shared memory in a controlled way, where each side maintains its view of the protocol state; all memory locations used to exchange information between the sides are allowed to be updated by a single designated writer (i.e., each shared memory location can be written by either the socket back-end or the socket-front-end within the appropriate application). Each application uses separate shared memory segment for writeable and readable parts of the queue state. Also, complete separation is maintained between the applications.

The design is somewhat similar to direct-access TCP offload solutions with interface comparable to Virtual Interface Architecture (VIA [3]), when the notification queues serve to emulate doorbells, and command/response queues are implemented as lock-free producer-consumer queues.

4.2 Socket Send Operation

One of the key issues in the design of efficient data transfer (for any type of I/O) is memory management. This issue is particularly complicated for communication services based on legacy, streaming-mode, synchronous socket API, due to inherent data copy semantics and unpredictable patterns of application operation. In particular, a large data transfer is likely to be implemented as a sequence of multiple smaller transfers, invoked synchronously, passing data residing at arbitrary locations. This observation, together with the fact that the data copying overhead becomes less pronounced on modern systems ([24]), underlies our decision to avoid zero-copy design for socket send operations – even though such a decision seems counter-intuitive, as zero-copy property is considered a holy-grail of network acceleration solutions. Zero-copy solutions tend to offer improved performance at the cost of application modification (e.g., through new asynchronous APIs), and are only beneficial for a subset of workloads. We, on the other hand, strive to improve performance for a broad range of existing unmodified applications. In particular, one of our design goals was to keep (or improve) the low latency of the synchronous send call. Thus, we chose to keep the single data copy, performed on the application side.

In our solution, the synchronous API is implemented using socket transmit buffers that are pre-allocated and pre-registered for the DMA access. This significantly reduces buffer management overhead and allows efficient aggregation of small data chunks. The socket back-end allocates DMA-able memory segments for each socket application; during socket initialization, the socket front-end (kernel helper) allocates per-socket transmit buffers out of the DMA-able chunk and maps them for user-space access. When the application sends data, the socket front-end copies the data from the application buffers into the socket transmit buffer (mapped into the application address space) used as a contiguous cyclic buffer. Afterwards, the socket front-end writes a transmit command to the socket command queue, specifying the location of new transmit data within the socket buffer. To simplify memory protection, it does not use pointers to identify the data in the transmit buffer, and instead uses offsets relative

to the buffer start. When the socket back-end receives the command, it uses the buffer registration information and the specified offset to construct the DMA address to be passed to the device driver. The socket back-end does not access the transmit buffers; it just serves as an intermediary that facilitates the buffer sharing between the socket front-end and the NIC.

The implementation of the send call copies the application data to the transmit buffer; the space occupied by the copied data is reused after the socket back-end reports that it was delivered to the remote receiver. The buffer space is used to facilitate the batching of multiple small requests in case the sender is faster than the local stack or the receiver. The socket front-end does not necessarily notify the socket back-end about each new piece of data that was copied to the transmit buffer. Instead, it aggregates data if the amount of previously posted pending data becomes high, until the socket back-end reports sufficient progress on the data transmission, or until a large amount of data has accumulated. Thus, the data aggregation does not increase latency; it occurs only when the previously submitted data starts piling up.

In turn, the socket back-end performs additional aggregation, postponing the TCP processing of newly submitted data when the TCP connection state does not allow immediate segment generation (i.e., when the TCP send window or congestion window is full). Like the aggregation at the socket front-end, the aggregation at the socket back-end does not introduce unnecessary delays; it decreases the TCP overhead and the overhead of the interaction with the device, due to better utilization of its TCP segmentation capabilities.

4.3 Socket Receive Operation

Handling incoming network traffic using a regular NIC is a known challenge. Due to unpredictable patterns of packet arrival, the packets received by a stateless NIC must land into anonymous buffers that are not associated with a particular connection. The packet data must be copied from the anonymous kernel buffers to the application buffers, which may be provided by the application after an arbitrary delay; thus, complex bookkeeping of the packet data structures is needed. The main design choice we had to make was the context for performing the data copy operation.

One choice would be asynchronous copy by the socket back-end, which seems to offload a maximal number of CPU cycles from the application CPUs. However, this approach has numerous drawbacks. It causes thrashing of the IsoStack resources such as cache, TLB, and SLB, and it may actually decrease the application performance due to increased latency of receive operation and decreased cache locality; this occurs when the application tries to access the newly received data, which was brought to the wrong cache during the copy. Accordingly, we decided to copy the data on the application CPU, within the socket front-end.

Applications (or their writers) expect the latency of the receive socket call to be very low if the data already arrived. In order to minimize this latency, our implementation strives to perform the copy during the synchronous execution of the receive call, without interacting with the socket back-end. To achieve that, the socket front-end "prefetches" receive buffers from the socket back-end in advance, independently of the receive calls invoked by the application, using asynchronous requests. Upon such request, the socket back-end hands over to the socket front-end the ownership on the data buffers that contain the receive data stream of the socket (when these are available). Multiple buffers corresponding to multiple network data segments can be reported at once, decreasing the interaction between the socket front-end and the back-end. If the previously posted request is completed before the application invokes socket receive function, socket receive implementation in the socket front-end copies the data immediately; otherwise, the application blocks until the previously requested data buffers are available. The socket front-end uses a heuristic to decide when to request more buffers.

Since the packet buffers reside in kernel space and cannot be mapped in advance to the relevant application, the receive pointers queue is maintained in the kernel by the socket helper kernel module, which also copies the data during the socket receive call invoked by the application. This necessitates a kernel boundary crossing upon each receive operation, thus incurring a higher overhead than the send. However, the overhead is still lower than the native implementation because the socket front-end state is only accessed locally, unlike the regular socket object in the native stack, which is shared between different stack components running in different contexts.

4.4 Event-Driven IsoStack Operation

The IsoStack is implemented as a single-threaded non-preemptive processing loop, serially handling asynchronous events. A dispatcher component of the IsoStack polls event queues to detect the new work to be done such as new packet arrivals, new application

requests to be executed, or timeout expiration; it then invokes appropriate event handlers sequentially. The device is configured to operate in polling mode; a new device driver entry point is used to poll periodically for new packet arrival. The message queue mechanism also allows periodic polling of the socket command queues (or, more precisely, event notifications queues). The polling is done by reading from a cache-coherent memory location, thus busy-wait polling on empty queues is inexpensive, because it is usually accomplished by access to the local cache only.

The socket back-end running within the IsoStack executes the commands delegated by the socket front-end. If it cannot execute a command immediately, it postpones the command execution until an appropriate change of state occurs (e.g., until incoming data is buffered, in the case of the receive command). Each such command is implemented as a separate state machine. For example, if the socket front-end is requested to send data on a socket when the transmit window is full, the command handler puts aside the command state and marks the socket to enable asynchronous notification when transmission becomes possible. It then returns, allowing the dispatcher to proceed with other work. When an ACK packet arrives on the appropriate connection, the adapter's polling receive handler (invoked by the dispatcher) passes the packet up the stack; the TCP processing layer performs its regular processing and then generates an internal event indicating that the window space is freed. Later, the dispatcher detects the internal event and passes it to the socket back-end, which resumes execution of the send command.

4.5 Lock Elimination

Our architecture allows elimination of locks that were introduced within the network stack as a part of support for multiprocessor systems. Since the socket back-end objects and the network interface data structures are accessed sequentially in the context of the IsoStack thread, there is no need to worry about mutually exclusive access for these resources, which are private to the IsoStack. We made minimal modifications to the appropriate stack components to bypass the locking/unlocking code when touching the device or socket resources that belong exclusively to the IsoStack.

Many other stack resources, such as the hash table of TCP connections or IP routing table, are shared across the system. To better utilize the advantages of our architecture, it is desirable to avoid this sharing and allow local-only access instead. These structures can be split into independent instances, each holding the relevant portion of information, potentially replicated and updated only using explicit "messages" delivered as internal events. For example, the generic Ethernet handling layer uses a lock to protect access to shared device configuration information that is changed rarely, if ever, using management interfaces. In our architecture, the IsoStack must be the exclusive owner of configuration information for the devices assigned to it; the management interfaces need to be intercepted, and execution of configuration changes need to be delegated to the IsoStack. This would make locking unnecessary, since the device configuration is accessed serially. Our experiments show that even uncontended locks incur a high overhead; thus, elimination of these remaining locks can yield an additional tangible performance improvement.

5. Experimental Results

This section demonstrates the performance improvement that can be achieved using the IsoStack approach. We use several micro-benchmarks to emulate different workloads, and evaluate the performance of several variants of the IsoStack, using the native (unmodified) stack as a baseline.

5.1 Experimental Setup

Our system under test is a Power6 machine, connected back-to-back to a "remote" system over a 10Gb/s link. Both machines have an additional NIC used for remote access. The Power6 system is a 4-way (8 core) system, running at 3.5 GHz, with 16 GB of RAM, equipped with a 10Gb/s HEA (Host Ethernet Adapter). All physical resources are assigned to a single logical partition (LPAR), which runs the AIX 6.1 operating system. Since the cores provide two-way SMT (symmetrical multithreading) capabilities, the machine appears to have 16 logical processors from the point of view of the OS. The remote system is a quad core AMD Opteron machine with 2GB RAM, equipped with 10G Broadcom NetXtreme II BCM57710 NIC, running Red Hat Enterprise Linux 5.3 (2.6.18 kernel).

Our experiments compare the AIX native TCP/IP stack with the IsoStack, using the same micro-benchmark applications. To measure the IsoStack performance, we ran the IsoStack socket back-end and the test applications linked with the socket front-end. To obtain AIX native results, we re-ran the same tests linked with the regular socket library over the unmodified AIX kernel and the unmodified network drivers with the

same adapter configuration parameters. To achieve maximum bandwidth (on both types of systems), we increased the dedicated interfaces' MTU to 9000, disabled hardware flow control, and enabled TCP checksum offload and TCP segmentation offload. The AIX built-in Nmon tool ([32]) was used to measure network throughput and CPU utilization.

In order to evaluate scalability of our implementation for multiple application threads, we used a multi-threaded TTCP-like application, where each thread sends or receives data over a single socket. We measured the achieved throughput, and the total CPU utilization for all processors (i.e., 100% means all cores are fully utilized; a single core accounts for 12.5%). Note that the IsoStack core is always fully consumed, because of polling-mode operation. CPU utilization of IsoStack shown below includes the constant utilization of the IsoStack core, and varying CPU utilization of the IsoStack socket front-end on application cores.

5.2 IsoStack Variants

To analyze the design choices, in particular those related to queuing and aggregation mechanisms, we implemented different variants of the IsoStack:

- Iso-Kernel. This implementation is described in Section 4. In particular, it supports transmit data aggregation, and uses in-kernel per-CPU notification queues; the socket back-end polls only the notification queues.
- Iso-Basic. Each application thread has a separate command/status queue in user-mode. No aggregation is used; each socket command translates to a message in the command queue. The socket back-end polls all the command queues.
- Iso-Aggregated. Uses the same queue structure as the Iso-Basic; implements client and server side transmit data aggregation.
- Iso-Lock. This variant is similar to Iso-Kernel; it reintroduces some of the locks that were eliminated in the other variants. The sole purpose of this variant is to evaluate the impact of un-contended locks, by an experiment described in Section 5.5.

The Iso-Kernel variant is the implementation that we used for most tests. In the rest of this section, unless stated otherwise, the term "IsoStack" refers to Iso-Kernel variant.

5.3 Throughput Evaluation

We used a multi-threaded TTCP-like application to evaluate basic data streaming. We measured the achieved throughput, and the total CPU utilization for all processors.

Since maximal throughput of a single connection is limited by end-to-end TCP behavior, the merit of IsoStack becomes more evident as more TCP connections are used. When the traffic amount is low, the socket back-end dedicated CPU is underutilized, and most of its cycles are wasted on polling empty queues. The observed results in many of the tests with low number of connections showed that the overall machine CPU utilization with the IsoStack implementation is higher compared to the native stack. However, when the number of connections starts to grow, this effect is quickly mitigated and the IsoStack shows not just an increased or identical bandwidth, but also lower CPU utilization.

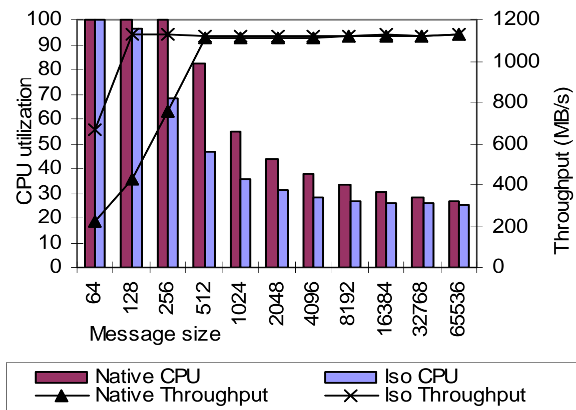

Figure 4. Receive performance for 64 connections

Figure 4 demonstrates receive performance for different message sizes for 64 connections (and 64 application threads). For small messages (64 bytes or 128 bytes), the IsoStack achieves bandwidth that is about 300% better than native, while both systems use almost all available CPU cycles. Clearly, CPU cycles are better used when CPUs are asymmetrically divided between the applications' CPUs and TCP. As message sizes increase, both stacks achieve the line speed with declining CPU utilization, although the native stack still uses more CPU cycles than the IsoStack to drive the same bandwidth. For message sizes above 16 KB, the performance improvement is less prominent: the throughput remains maximal for both stacks, CPU utilization of the IsoStack appears constant (although in fact the dedicated CPU spends more time in polling

empty queues), and the CPU utilization of the native stack decreases, as there are fewer system calls for the same amount of data.

Figure 5 demonstrates the transmit performance for different message sizes using 128 connections. The IsoStack reaches the line speed even for a message size as small as 64 bytes, whereas the native stack can reach the line speed only for message sizes of 16 KB and above. Moreover, the IsoStack utilizes far fewer CPU cycles than the native stack. The difference is more dramatic for small messages, where the native stack uses 200% more CPU cycles (while driving a fraction of throughput) than the IsoStack. However, the difference is still high even for large message sizes, when both stacks achieve close to line-speed throughput and the native stack consumes 50% more CPU cycles when compared to the IsoStack.

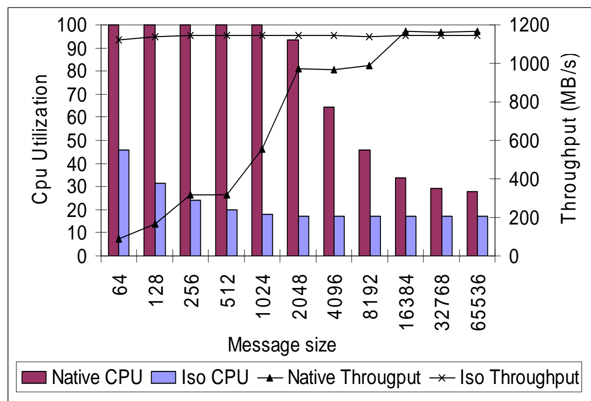

Figure 5. Transmit performance for 128 connections

5.4 Request-Response Performance

In this section, we discuss performance of request/response workloads. Each of the test application threads repeatedly sends and receives a single message, simulating typical client-server communication pattern. This type of workload maximizes the overhead for delegating socket operations to the IsoStack, since each socket operation involves interaction with the stack as no aggregation is taking place.

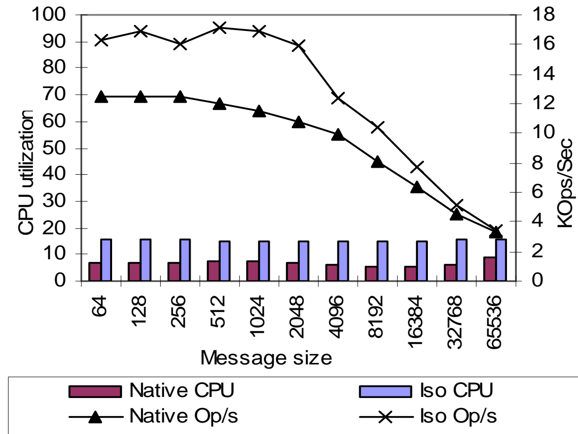

Figure 6. Request-Response test, one connection

Figure 6 demonstrates the request-response test performance for different message sizes using one connection. This allowed us to focus on the impact of socket delegation, without any additional improvements due to aggregation or reduced contention. In this scenario, the IsoStack provides more operations per second for all message sizes, although the difference between the stacks diminishes as the message size increases. Thus, the average latency of a single request-response transaction improves when the IsoStack is used, which may seem surprising because of the added latency imposed by interaction between socket front-end and socket back-end. However, this additional latency of socket delegation is offset by the decreased latency of the network processing, due to lock-free and interrupt-free operation.

Because of the synchronous nature of this test (with just one operation in-flight), the performance is very low for both stacks, due to the delay caused by waiting to the remote application. The CPU utilization for the IsoStack appears to be higher than that of the native stack, since the socket server CPU – although underutilized – still uses 100% of its resources due to wasted polling cycles.

To test the system scalability under the request/response workload, we ran the request-response test with varying numbers of connections (or, equivalently, application threads). Figure 7 shows the CPU utilization and the number of operations per second of both native stack and IsoStack, for different connection numbers, using a message size of 1KB. For up to eight connections, the native stack and IsoStack achieve a similar number of operations per second. For a higher number of connections, the IsoStack CPU becomes fully utilized, and turns into a bottleneck. The native stack allows multiple threads to utilize all processors in the system,

and each socket call is executed immediately, even if relatively slowly, on the calling processor. On the other hand, the IsoStack forces serialized execution of socket operations invoked for different sockets on different processors, and thus induces a queuing delay when many processors submit their operations in parallel. Thus, the native stack is able to make progress on each connection faster than the IsoStack, even though its CPU utilization per operation is higher.

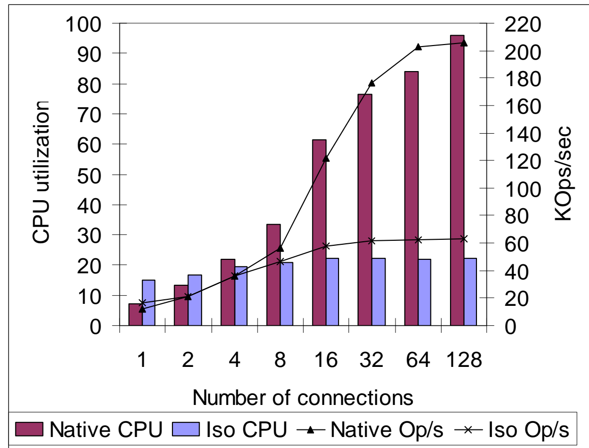

Figure 7. Request-Response test, 1KByte messages

To analyze further the bottleneck imposed by the IsoStack, we measured various code paths inside the socket back-end CPU. We found that simply issuing the kernel call that wakes up the socket application (waiting to receive data) takes approximately 3µs. To compare, the optimized TCP send operation (involving TCP, IP, and MAC layers) also takes approximately 3µs, the socket back-end operation (without the wakeup) takes less than 1µs, and the whole request/response transaction accounts for approximately 16µs. Analysis of the wakeup call shows that the problem is mainly due to contention on several scheduler locks. This indicates that the IsoStack performance could be improved further if a more efficient wakeup mechanism is used.

5.5 Impact of Uncontended Locks

It is a popular belief that reducing lock contention is sufficient to address the problem of the lock overhead. Our implementation went one step further, and eliminated some of the locks completely, avoiding the lock operations altogether for the locks that are only taken on the IsoStack processor. To evaluate the impact of this optimization, we tested an additional variant of the IsoStack, called Iso-Lock, in which we re-instantiated some of the locks – even though they are not needed in our architecture and are only accessed by the IsoStack CPU.

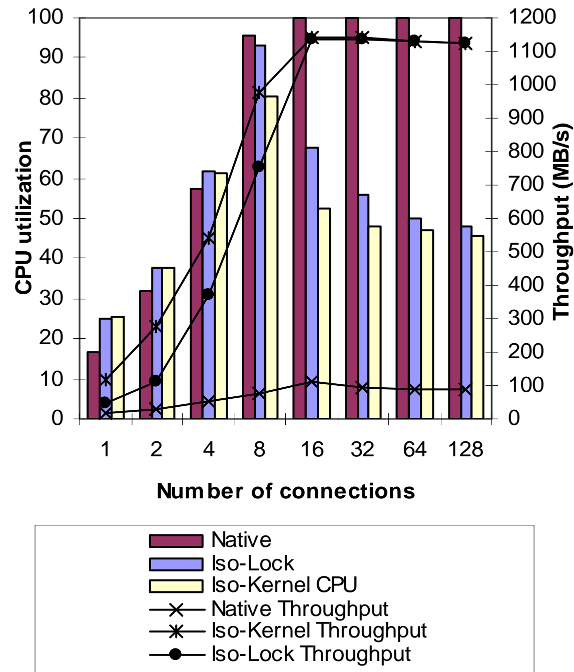

Figure 8. Impact of extra lock on transmit performance for 64 byte messages

Figure 8 depicts the effect that re-instantiating the locks had on the IsoStack performance. For this experiment, we re-introduced the HEA device driver TX and RX locks. These locks were acquired and released each time the device driver transmit or receive handler were called. Socket send throughput tests were performed with a fixed message size of 64 bytes and a variable number of connections. We used the native stack results as the baseline. For a small number of connections, the IsoStack achieves superior throughput compared to the Iso-Lock version, while the CPU utilization appears to be the same. The throughput improvement due to the eliminated lock reaches 200MB/s for eight connections. As the connection number increases, both implementations reach line-speed. The CPU utilization of Iso-Lock is higher than the regular IsoStack variant, which means, oddly, that the socket front-end consumes more CPU. This stems from the fact that additional locks (even though uncontended) make the socket back-end CPU perform slower; the socket transmit buffers then fill up more frequently, causing the socket front-end to wait for free space in the TX buffer. As a result, additional CPU cycles are spent on the extra scheduling that is involved in waking up the socket front-end.

This experiment shows clearly that even un-contended locks are a significant source of overhead. This result may seem counterintuitive, as kernel lock implementation usually takes just a few instructions. Indeed, the locking instruction path length is short, and the atomic update instructions are cache-hits. However, the lock implementation is also required to use a memory barrier – *heavy-weight sync* instruction ([34]), which causes long CPU stall.

Since our implementation did not eliminate all locks that became redundant, the remaining locks pose potential for additional improvement.

5.6 Evaluating Different Queuing Mechanisms

In this section, we try to analyze the performance of queuing mechanisms implemented in the different IsoStack variants.

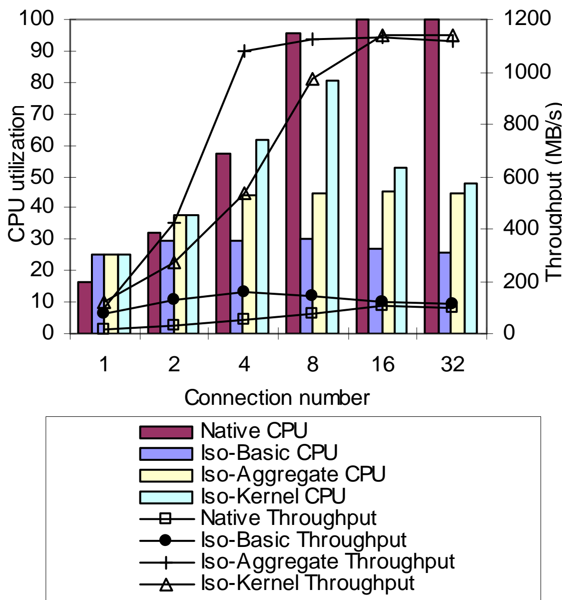

Figure 9. Transmit performance for three IsoStack variants, 64 byte messages

In Figure 9, we compare the 64-byte transmit performance of Iso-Basic (per thread notification queues without aggregation), Iso-Aggregate (per-thread notification queues with aggregation of transmit operations) and Iso-Kernel (per-CPU notification threads with transmit aggregation), with the native stack as a baseline. All three IsoStack variants achieve better throughput with reduced CPU utilization, compared to the native stack. Iso-Aggregate and Iso-Kernel achieve up to eleven times (1000%) more bandwidth than the other variants due to the aggregation that both employ. As a result, they both use more CPU than Iso-Basic, although they still use remarkably less CPU than the native stack. Due to the high cost of using the kernel notification queues, Iso-Aggregate performs better than Iso-Kernel for a low number of connections, but as the number of connections (and application threads) grows, Iso-Aggregate throughput declines, while Iso-Kernel stays at the same throughput with decreased CPU utilization, and eventually out-performs the Iso-Aggregate.

The scalability advantages of the Iso-Kernel variant can be seen more clearly in Figure 10, which depicts the results of a request-response test for varying numbers of connections. The performance of Iso-Aggregate drops dramatically as the number of connections grows beyond 16, while the Iso-Kernel stack scales gracefully, i.e., increased number of clients does not cause performance degradation. This is due to the reduced polling overhead for the socket back-end in the Iso-Kernel implementation, as it polls only the constant number of notification queues, unlike the Iso-Aggregate variant that polls a separate queue for each application thread.

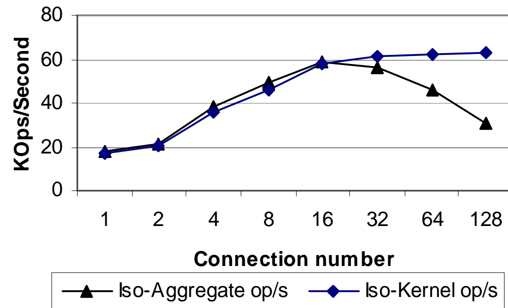

Figure 10. Request/Response Scalabilty

6. Conclusions and Future Work

Our work shows that the design principles of asynchronous interaction, non-shared state, and non-shared processor resources for demanding tasks can be applied to network stack design, yielding significant performance improvements for most workloads. However, some workloads remain challenging. For example, we encountered scenarios where the serialized execution within the IsoStack introduces additional latency when processing particular events. The dispatching of network events handling is rather unsophisticated in our implementation, where the basic policy arbitration is weighted round-robin between the different event queues. Other arbitration policies need

to be evaluated, possibly involving a real-time scheduler. Also, it would be beneficial to identify latency-sensitive flows (automatically or with the help of application-provided quality-of-service hints), and prioritize their handling.

We evaluated the system performance for a 10 Gb/s network port, using a single dedicated processor core. As network speed continues to grow, with emerging support for 40 Gb/s and 100 Gb/s, while the processor speed is not expected to increase, it will soon become necessary to employ multiple cores to handle network traffic for a single port in parallel. Fortunately, multi-queue support and minimal packet classification capabilities, available in state-of-the-art adapters, allow parallelization of network processing without re-introducing dependencies between the processors. The IsoStack can be parallelized using independent stack instances for disjoint subsets of network flows, using separate control data structures, and interacting with the client applications through distinct queues.

Our experience shows that dedicating processor cores to specific tasks can improve the overall system performance and scalability. However, the performance gains come at a price: a significant development effort is needed to integrate "isolated" components successfully within a system that was designed under a completely different paradigm. Our implementation had to refrain from using existing system services, as they brought back the very problems we were trying to solve. We believe these services should not be re-invented for every subsystem that can benefit from isolation; instead, the operating system should provide adequate support for isolated execution. Moreover, the underlying hardware should provide better support for inter-processor communication within the system, to supply a better infrastructure for subsystem isolation.

The implementation described in this paper addresses a single OS environment. However, one of the original goals of this work was to devise an architecture for efficient network virtualization. The general architecture described in [26] allows multiple clients to share an isolated I/O subsystem which runs on a different physical machine in a cluster environment or on a different virtual machine within the same physical system. Ironically, interaction between physical machines over a cluster interconnect turned out to be more efficient than interaction between virtual machines within the same POWER system. To realize the performance potential of the IsoStack for virtualized systems, the hypervisor and the underlying hardware have to provide better support for efficient inter-processor communication between processors assigned to different virtual machines.

7. Acknowledgements

We would like to thank Pratap Pattnaik for the idea to evaluate the IsoStack architecture in AIX operating system. Additionally, we thank Joefon Jann, R. S. Burugula, Tom Mathews, Venkat Venkatsubra, G Shantala, Rakesh Sharma and Dave Marquardt for helpful discussions and for making it possible for us to use the AIX development environment. We also thank Herman Dierks for inspiring discussions on performance evaluation, Alan Jiang for his expert input on AIX services, and Shay Goikhman for his participation in the implementation efforts.

8. References

[1] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. IEEE Communications Magazine, 27(6):23–29, June 1989.

[2] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols, pages 259–268. ACM, September 1993.

[3] P. Buonadonna, A. Geweke, D. Culler. An Implementation and Analysis of the Virtual Interface Architecture, In Proceedings of SuperComputing '98.

[4] S. Muir and J. Smith. Functional divisions in the Piglet multiprocessor operating system. In Eighth ACM SIGOPS European Workshop, September 1998.

[5] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. IEEE Communications, Special Issue on High-Speed TCP, 39(4):68–74, April 2001.

[6] E. P. Markatos. Speeding-up TCP/IP: faster processors are not enough. In Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference (IPCCC 2002), April 2002, pages 341-345.

[7] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, W. Zwaenepoel. TCP Servers: Offloading TCP Processing in Internet Servers–Design, Implementation and Performance. Rutgers University Department of CS TR, DCS-TR-481, 2002.

[8] P. Buonadonna and D. Culler. Queue-pair IP: A hybrid architecture for system area networks. In

Proc. 29th Ann. Int'l Symp. on Computer Architecture, pages 247--256, May 2002.

[9] The Infiniband Trade Association. The Infiniband Architecture. http://www.infinibandta.org/specs.

[10] A. Romanow, and S. Bailey. An Overview of RDMA over IP. In p roceedings of the First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003), Feburary 2003.

[11] J. Mogul. TCP offload is a dumb idea whose time has come. In Workshop on Hot Topics in Operating Systems (HotOS). May 2003.

[12] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When does hardware support help? In 2nd USENIX Symposium on File and Storage Technologies (FAST), March 2003.

[13] P. Shivam, J. S. Chase. Promises and reality: On the elusive benefits of protocol offload. In ACM SigComm Workshop on Network-IO Convergence (NICELI), 2003.

[14] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, A. Foong: ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. A Symposium on High Performance Interconnects (HOT Interconnects), 2003.

[15] D. McAuley and R. Neugebauer. A case for Virtual Channel Processors. In Proceedings of the First Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI), 2003.

[16] A. Foong, T. Huff, H. Hum, J. Patwardhan. TCP Performance Re-Visited. In Proc. 2003 IEEE Int'l Symp. Performance Analysis of Systems and Software (IPASS 03), 2003, pp. 70-79.

[17] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. IEEE Computer, 37(11):48--58, November 2004.

[18] ‘‘Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS,’’ white paper,WinHEC 2004,Microsoft.

[19] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In USENIX Annual Technical Conference, April 2005.

[20] V. Saletore, P. Stillwell Jr, J. Wiegert, P. Cayton, J. Gray, G. Regnier. Efficient Direct User Level Sockets for an Intel® Xeon™ Processor Based TCP On-Load Engine. In Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, Apr. 2005.

[21] L. Grossman. Large Receive Offload Implementation in Neterion 10 GbE Ethernet Driver. Ottawa Linux Symposium, 2005.

[22] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, I. Shimony. Loosely Coupled TCP Acceleration Architecture. In Proceedings of 14th IEEE Symposium on High-Performance Interconnects (HOTI'06), Aug. 2006.

[23] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In Proceedings of Usenix Annual Technical Conference, June 2006.

[24] S. Larsen, P. Sarangam, R. Huggahalli. Architectural Breakdown of End-to-End Latency in a TCP/IP Network. International Symposium on Computer Architecture and High Performance Computing, 2007.

[25] A. Menon , W. Zwaenepoel, Optimizing TCP receive performance, USENIX 2008 ATC, p.85-98, June 2008.

[26] J. Satran, L. Shalev, M. Ben-Yehuda, Z. Machulsky. Scalable I/O - A Well-Architected Way to Do Scalable, Secure and Virtualized I/O. In Proceedings of Workshop on I/O Virtualization, 2008.

[27] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, p. 43–57, Dec. 2008.

[28] J. Liu, B. Abali. Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. ICS 2009: 225-234

[29] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In Proc. ACM Symposium on OS Principles, Oct. 2009.

[30] LPAR, http://en.wikipedia.org/wiki/LPAR, retrieved on December 22, 2009.

[31] Large segment offload, http://en.wikipedia.org/wiki/Large_segment_offload, retrieved on December 22, 2009.

[32] Nmon, http://en.wikipedia.org/wiki/Nmon, retrieved on December 22, 2009.

[33] Extended Sockets API, www.opengroup.org, retrieved on December 22, 2009.

[34] Power ISA, http://www.power.org/home.