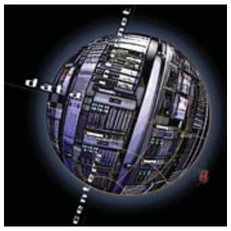
COVER FEATURE

TCP Onloading for Data Center Servers



To meet the increasing networking needs of server workloads, servers are starting to offload packet processing to peripheral devices to achieve TCP/IP acceleration. Researchers at Intel Labs are experimenting with alternative solutions that improve the server's ability to process TCP/IP packets efficiently and at very high rates.

Greg Regnier
Srihari Makineni
Ramesh Illikkal
Ravi Iyer
Dave Minturn
Ram Huggahalli
Don Newell
Linda Cline
Annie Foong
Intel

The Internet and the applications it enables have transformed data center architectures from a simple single-server model to a multitier model in which Web servers, application servers, databases, and storage servers work together to respond to each client's request. As a result, overall application performance has become more dependent on the efficiency of data center communication.

The Transmission Control Protocol is the most widely used protocol for reliable data communications both on the Internet and in data center networks. TCP, the transport layer in the four-layer Internet protocol suite, always runs over the Internet Protocol. IP, the network layer, provides routing services between source and destination machines while TCP provides guaranteed and orderly delivery of data to applications running at each end of the communication pipe. These two protocols are collectively referred to as TCP/IP (TCP over IP). First used in the mid-1980s, TCP/IP over Ethernet has increased in popularity with the evolution of the Internet and related applications such as Web browsing, e-commerce, e-trading, and messaging.

Inside data centers and corporate local area networks, TCP/IP runs on top of Ethernet, which provides, among other things, framing services (marking packet boundaries) to upper layer protocols and sends and receives bits on the wire. The de facto protocol on LANs, Ethernet is also being considered for metro and wide area networks.

Until a few years ago, Ethernet speeds inside data centers averaged 100 Mbps. However, increased Internet usage created data center networking demands that spurred the IEEE Ethernet Standards Committee's 802.3 subgroup to develop faster Ethernet network technologies. This work led to the development of 1 Gbps and 10 Gbps Ethernet networks. Today, 1-Gbps Ethernet networks are being widely deployed, and 10 Gbps will be widely deployed as it becomes affordable. The sudden jump in Ethernet speeds from 100 Mbps to 1 and 10 Gbps requires TCP/IP processing on the data center servers to scale proportionately so that network-intensive applications can ultimately benefit from the increased network bandwidth levels.

TCP/IP stacks traditionally have been implemented in software as part of the operating system kernel. TCP/IP interacts with the network interface controller (NIC) through a device driver and is exposed to applications through the traditional sockets interface. TCP/IP processing starts at the NIC hardware and extends all the way to the TCP/IP stack interactions with the application layer.

Our recent measurements on state-of-the-art platforms show that TCP/IP processing of application data—a full frame payload size of 1,460 bytes—consumes one entire current generation CPU to achieve roughly 750 Mbps of throughput while receiving data and around 1 Gbps of throughput while transmitting data. Clearly, it currently is not possible to achieve a tenfold increase in TCP/IP throughput. The major impediment to achieving 10-Gbps throughput is that CPU and memory speeds are not expected to improve as dramatically as Ethernet technologies.

While TCP/IP acceleration efforts focus on enabling 10-Gbps communication rates, improving the efficiency at which this rate is achieved is equally important. In a recent study of data center servers,[1] we analyzed the current and future networking requirements of Web servers, front-end servers, and database servers. This study showed that TCP/IP receive-and-transmit overhead consumed about one-third of the compute resources. Another recent study[2] showed that the processing overhead can be as high as 60 to 70 percent for Web servers after adding other TCP/IP overheads like connection setup.

To avoid the seriously limiting scalability issues of these server workloads in the future, TCP/IP processing efficiency must improve dramatically. Further, as different types of applications—some throughput-intensive, others latency-sensitive—employ TCP/IP, accelerating and scaling TCP/IP processing requires a balanced approach.

TCP/IP PROCESSING OVERVIEW

TCP/IP processing can be classified into three major categories:

- connection processing,
- transmit-side processing, and
- receive-side processing.

Connection processing refers to the establishment and teardown of TCP connections between communicating end systems. Once the systems establish a connection, data transfer occurs over the connection through receive and transmit operations. Typically, the frequent processing paths in data center servers are receive and transmit operations—also referred to collectively as data-path processing.

Receive-side processing

Receive-side processing begins when the NIC hardware receives an Ethernet frame from the network. To extract the packet embedded inside the frame, the NIC removes the frame delineation bits and updates the descriptor data structure with the packet information. The NIC driver software supplies these descriptors, which are typically organized in circular rings, to the NIC. Through these descriptors, the NIC driver informs the NIC of information such as the memory buffer address in which to store the incoming packet data. The NIC uses a DMA operation to copy the incoming data into this memory buffer. Once it places the packet in memory, the NIC updates a status field inside the descriptor to indicate to the driver that this descriptor holds a valid packet. It then generates an interrupt to initiate processing of the received packet.

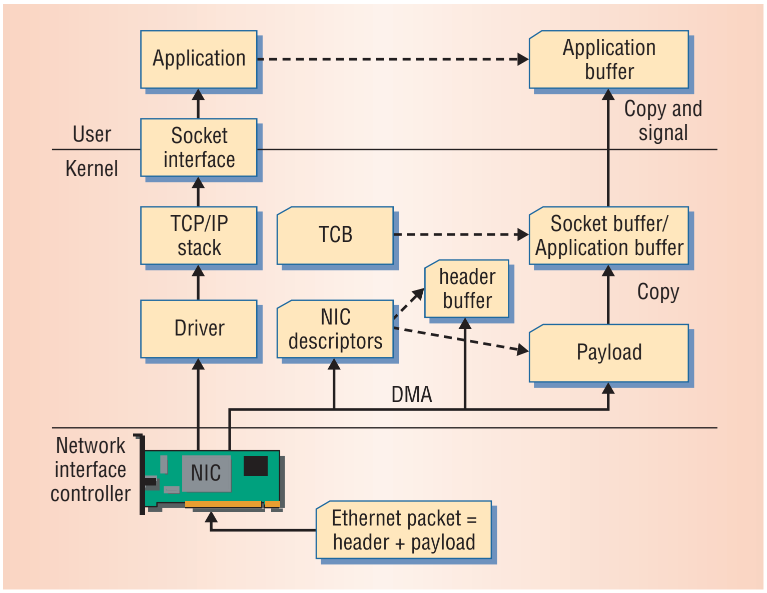


Figure 1. Data flow in receive-side processing. Receive-side processing starts with the NIC device driver reading the descriptor and ends with the TCP/IP stack notifying the application that the data has been placed in its buffer.

Figure 1 shows the overall receive-side processing flow. The NIC device driver reads the descriptor and passes the header and payload memory locations to the TCP/IP stack.

The next step is to identify the connection that this packet belongs to. The TCP/IP software stores each connection's state information in the TCP/IP control block (TCB) data structure. Since there can be many open connections and thus many TCBs, the TCP/IP stack uses a hashing mechanism to perform a fast lookup of the right TCB. The hash value is calculated from the IP address and port number of both the source and destination machines.

When the TCP/IP stack receives a new packet, it updates several fields in the TCB—such as the sequence numbers for received and acknowledged bytes—and checks whether the application has preposted buffers. If the application has already posted a buffer to receive the incoming data, the TCP/IP stack copies the incoming data directly from the NIC buffer to the application buffer. Otherwise the stack stores the data in a temporary buffer for later delivery to the application. Performing a memory copy to move data from the NIC buffer to the application buffer is one of the most time-consuming operations in receive-side processing.

Transmit-side processing

Transmit-side processing starts when an application passes a data buffer to the TCP/IP stack. The application passes a socket ID along with the data buffer, and the TCP/IP stack uses this socket ID to locate the connection's TCB. The TCP/IP stack can then copy the application's data into an internal buffer. Many TCP/IP stacks employ optimizations to avoid this copy. When the receiver's window size indicates it is time to transmit data, the TCP/IP stack divides the accumulated data into maximum transfer unit segments. An MTU is typically 1,460 bytes on Ethernet LANs. The stack then computes the header: 20 bytes for TCP, assuming no options, and 20 bytes for IPv4. The stack then appends these segments with the IP headers and passes them down to the NIC driver. The driver sets up the DMA to transfer the headers and application data to the NIC.

The imminent need to achieve 10 Gbps is forcing researchers to revisit TCP/IP overheads and challenges.

TCP/IP PROCESSING EVOLUTION

Over the years, applications with differing requirements for latency sensitivity and high throughput have adopted TCP/IP over varying link speeds. To make TCP work best across these various scenarios, researchers have investigated several enhancements to the base protocol.[3]

Initially, the primary source of overhead in TCP/IP processing was unclear, and developers assumed that the base protocol processing was the culprit. However, landmark research analyzing the associated overheads showed that TCP/IP protocol-specific processing by itself is not the main overhead.[4] The source for most of the overhead was the environment—interrupts, OS scheduling, buffering and data movement—in which the TCP/IP protocol operates. While this dissuaded researchers from replacing the TCP/IP protocol, overall processing continued to be expensive.

To ease some of these overheads, researchers developed several mechanisms that have become common in today's platforms and TCP/IP stacks:

- *Interrupt coalescing*. Also referred to as *interrupt moderation*, this mechanism helps reduce the overhead of using the interrupt mechanism to signal the arrival of incoming packets from the network. To amortize interrupt cost, NICs can accumulate multiple packets and notify the processor only once for several packets.
- *Checksum offload*. TCP and IP headers use checksums to ensure that the packet data is not corrupted during transmission. Checksum calculation on a general-purpose processor is an expensive operation, but TCP/IP stacks can offload this feature to the NIC if it supports checksum calculation. Since implementing a checksum in hardware is relatively simple, offloading it to the NIC hardware does not add much complexity or cost.
- *Large segment offload* (LSO). Segmenting large chunks of data into smaller segments and computing TCP and IP header information for each segment is expensive if done in software. Most current-generation NICs support this feature because doing this in hardware does not add much complexity. This feature benefits only transmit-side processing and is only beneficial when the application wants to send data larger than the maximum segment size (MSS), which the connection's two end points negotiated during the TCP establishment phase. When transmitting a 64-Kbyte application payload, LSO can improve performance by up to 50 percent.

Combined with constantly increasing CPU and memory speeds, these enhancements have enabled server platforms to achieve TCP/IP transmit throughput that meets or exceeds current Ethernet speeds (10-1000 Mbps) with reasonable efficiency. However, the imminent need to achieve 10 Gbps for both receive and transmit processing.

SERVER PROCESSING CHALLENGES

Today's server platforms face major challenges while performing TCP/IP processing.

Figure 2 shows an example of the processing breakdown for a TCP/IP transmit of 1-Kbyte packets. This measurement was taken on an Intel Xeon processor-based server system running Linux OS v2.4.18. In this example, OS integration and system overhead—interrupt, syscall, driver, and bufmgt components—constitute 50 percent of the overall TCP/IP processing. This creates significant overhead because the server's TCP/IP stack must share the CPU, memory, and I/O resources with the OS. To do this, the TCP/IP stack must follow the OS's rules in terms of scheduling, resource management, and interfaces.

OS mechanisms such as system calls, preemptive scheduling, layered drivers, and interrupt processing are all sources of overhead that limit TCP/IP efficiency and performance on servers. Researchers are investigating hybrid interrupt and polling mechanisms such as NAPI[5] to avoid the overheads and harmful effects of interrupts.[6]

The second major challenge for TCP/IP processing on servers is memory latency and poor cache locality. Server memory and cache architectures are optimized for efficient execution of a broad set of applications. The TCP/IP software stack generally exhibits poor cache locality largely because it deals with large amounts of control information and data that is entering or exiting the system through the network interface. For example, once the NIC receives packets and places them in memory, the TCP/IP software must examine the control information that includes the descriptor and packet headers for each packet. Since the packet was just placed in memory, access to this data will always cause a cache miss and thus a CPU stall.

Incoming application data also exhibits no temporal locality because the network interface places the incoming data into memory and forces the CPU to suffer several cache misses—that is, memory accesses—to bring the data into the cache for processing. I/O-intensive workloads cause numerous CPU stalls due to cache misses.[7] Given the rapid advances in network link technologies combined with the growing disparity between processor speeds and memory latency,[8] this is becoming a greater problem over time.

Buffering and copying application data offers the third major challenge in TCP/IP processing on servers. The TCP/IP stack copies incoming data into internal buffer space if the application has not posted any buffers to receive the data. The stack will copy the data again into the application buffer when it is available. Thus, depending on the scenario, there may be one or two copies of the incoming data.

When transmitting data, there is an opportunity to avoid a copy by transferring the data directly from the application buffer to the NIC using DMA. Zero copy on transmit has been achieved under some implementations—for example, in-kernel Web servers and fast-path overlapped I/O in the Microsoft Windows OS. Implementations based on traditional BSD sockets, however, still require a copy between application and kernel buffers in the general case. These data copies are expensive, especially when any of the source or destination buffers are not in the processor's cache.

While efforts have been made to eliminate copies, the proposed techniques are not applicable in many scenarios. Even if we assume that it somehow alleviates system overheads, the significant amount of time spent on memory copies and other memory stalls during TCP/IP receive-side processing can limit the achievable throughput to much less than 10 Gbps.

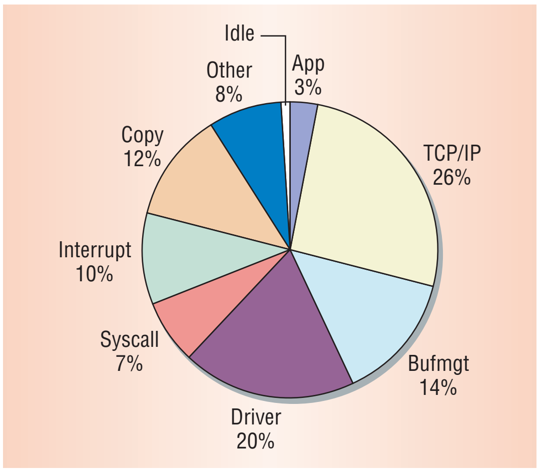


Figure 2. Profile of TCP/IP processing on servers. OS integration and system overhead constitute 50 percent of the overall TCP/IP processing.

TCP/IP ACCELERATION SOLUTION

To achieve efficient TCP/IP throughput of 10 Gbps and beyond on server systems, researchers and industry projects have adopted an approach that offloads the bulk of TCP/IP processing from server processors and executes it on a peripheral device. However, there is an ongoing debate about whether offloading TCP/IP processing is the right long-term solution.[10]

Offload or not?

TCP/IP offload engines (TOEs) generally offload the TCP/IP processing onto a device that attaches to the server's I/O system and uses separate, specialized processing and memory resources (www.alacritech.com/html/tech_review.html).

The main argument for TCP/IP offloading is that it increases the server's network throughput while reducing CPU utilization. For some application scenarios, especially for bulk transfers involving few connections, using a TOE can improve throughput and utilization for applications such as IP-based data storage.[10]

The arguments against offloading TCP/IP include factors such as scalability, flexibility, extensibility, and cost. In terms of performance, TOE devices generally have limited processing power because

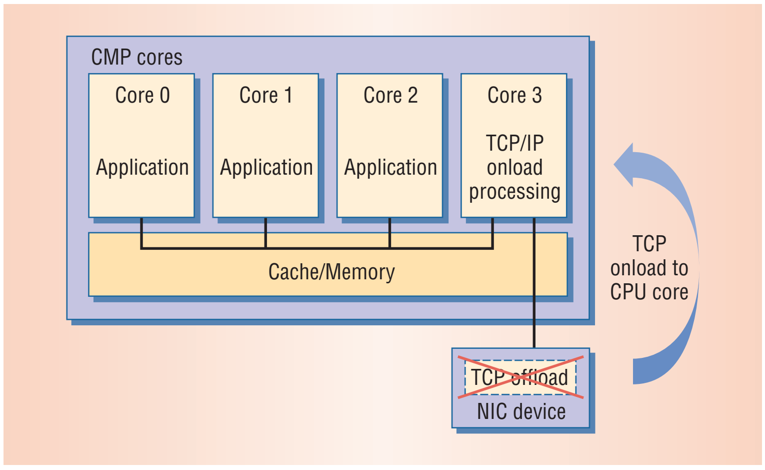


Figure 3. Onloading TCP/IP in a CMP server architecture. Using one or more of the cores efficiently for TCP/IP processing is preferable to offloading the processing to yet another compute element in a peripheral device.

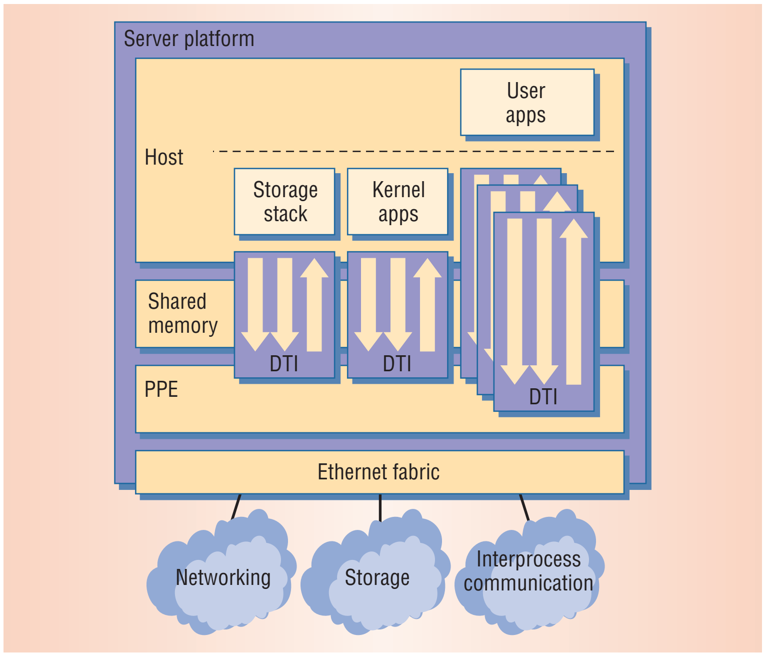


Figure 4. ETA architecture. The architecture partitions the system between the host and the packet-processing engine. The direct transport interface supports direct commands for establishing socket connections—connect, listen, accept. The interface also supports anonymous buffer pools for out-of-order packets and for incoming data without a corresponding posted receive buffer.

Moore's law applies to high-volume, general-purpose processors, and the processors in TOE devices tend to lag behind. Because most TOE devices must store and forward the payload data, the additional latency tends to hamper performance for real-life applications other than bulk data transfers.

Since TOE devices attach to the server platform's I/O subsystem, the latency to main memory is relatively high compared to the main CPU. Because of this additional latency, TOEs require local memory to store state and to buffer data. This local memory adds latency and cost, and the amount of memory constrains scalability in terms of how many concurrent connections a TOE can support.

The alternative to offloading is to improve the server's inherent ability to process TCP/IP packets efficiently and at very high rates, referred to as *onloading*. General-purpose processors, with their economies of scale, have the advantage of being flexible, extensible, and scalable. Their programmability and the availability of a rich set of programming tools make them extensible and flexible. Extensibility makes it possible to add value to the solution over time in terms of new features, protocols, and applications, while flexibility lets the networking software adapt to changing standards and modify its behavior in subtle but important ways.

Stack and NIC enhancements

Stack and NIC enhancements that are either already available or likely to become available in the near future to help server platforms achieve more efficient network throughput include the following:

- *Asynchronous I/O*. This technique lets applications use a completion port for receiving socket notifications. To reduce the number of copies needed during processing, applications can pre-post one or more buffers to send and receive data from the network and wait for completion messages to arrive in the completion port.
- *Header splitting*. Instead of placing the entire packet in one location in memory, this technique lets the NIC place headers independently from the data. In addition to allowing locality of incoming headers, this soon to be available technique facilitates better prefetching schemes.
- *Receive-side scaling*. This approach allows multiprocessor systems to better process network traffic on multiple CPUs (www.microsoft.com/whdc/device/network/NDIS_RSS.mspx). It accomplishes receiver-side scaling by providing multiple receive queues in the NIC and letting them be mapped to different processors in the system. This allows for scalability through connection-level parallelism and affinity. We anticipate that NIC vendors will employ this feature in future products.

While these enhancements definitely help improve the efficiency of TCP/IP processing, they are not adequate to scale TCP/IP processing to 10Gbps and beyond.

ONLOADING TCP/IP IN FUTURE SERVER PLATFORMS

The evolution of processor architectures is at an important juncture: As more transistors become available to processor architects, innovative architectures such as on-chip multiprocessors are becoming a reality. In CMP architectures, each processor in the server platform consists of multiple cores, a potentially shared last-level cache, and integrated memory support. As Figure 3 shows, in such an architecture, using one or more of the cores efficiently for TCP/IP processing is preferable to offloading the processing to yet another compute element in a peripheral device.

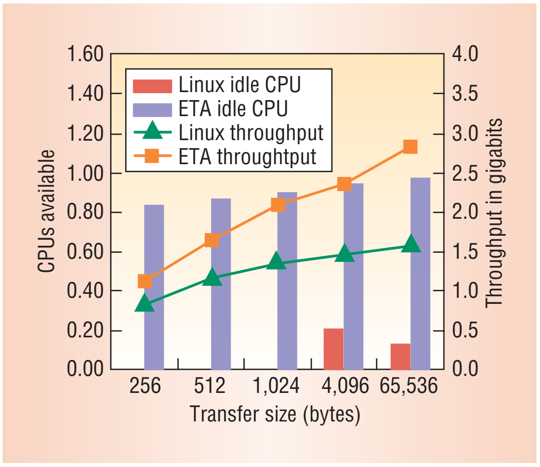


Figure 5. ETA transmit performance benefits. The ETA transmit throughput considerably exceeded the standard Linux server for all transfer sizes.

Addressing system overheads

To enable high-performance communication for servers over standard Ethernet and TCP/IP networks, the Embedded Transport Acceleration (ETA) project at Intel Labs reduces the operating system overhead by dedicating one or more of the cores for TCP/IP processing.[11] *Packet processing engine* (PPE) is the generic term we use for the cores that are dedicated as communication processing cores.

Figure 4 shows the high-level ETA architecture, which partitions the system between the host and the PPE. The system implements the interfaces between these two components in the shared host memory. Each direct transport Interface (DTI) consists of a set of queues for control, data, and synchronization.

The DTI is modeled after the virtual interface architecture[12] and Infiniband (www.infinibandta.org) interfaces, but has been optimized for TCP and socket semantics. In particular, the DTI supports direct commands for establishing socket connections—connect, listen, accept. It also supports anonymous buffer pools for out-of-order packets and for incoming data without a corresponding posted receive buffer.

We have implemented an ETA prototype by employing one of the processors in a dual-processor SMP server as the host processor and the other as the PPE. The PPE establishes and terminates TCP/IP sessions on behalf of applications running on the host CPU. The processing core uses multiple standard gigabit Ethernet network cards with a modified version of the Ethernet driver. In our testing, we used a dual-processor Intel Xeon processor platform. The processors run at 2.4 GHz on a 400-MHz front-side bus.

Our test environment consists of the server under test—the ETA prototype server—and five client computers connected directly by gigabit Ethernet links. The client computers are standard off-the-shelf servers running the Linux OS and the Test Transmission Control Protocol throughput microbenchmark. The tests running on the ETA prototype are kernel-level applications that interface directly to an ETA kernel abstraction layer. We performed basic throughput tests on the ETA prototype for transmit and receive for several transfer sizes, then we compared the initial ETA test results with a standard Linux dual-processor server running the TTCP benchmark.

Figure 5 compares transmit performance along with the amount of idle CPU processing time available for application use. For transfers of 1,024 bytes and less, both CPUs in the standard Linux server were 100 percent utilized executing the networking stack, thus leaving no CPUs idle. For larger-sized messages, 20 percent or less of one CPU was left idle. For the ETA prototype, the host CPU was used less than 20 percent across all transfer sizes, leaving more than 80 percent of one CPU idle and available. In addition, the ETA transmit throughput considerably exceeded the standard Linux server for all transfer sizes.

Figure 6 shows a comparison of the efficiency of the standard Linux (SMP) server and the ETA prototype in terms of the bits/hertz rule-of-thumb for both transmit and receive tests.

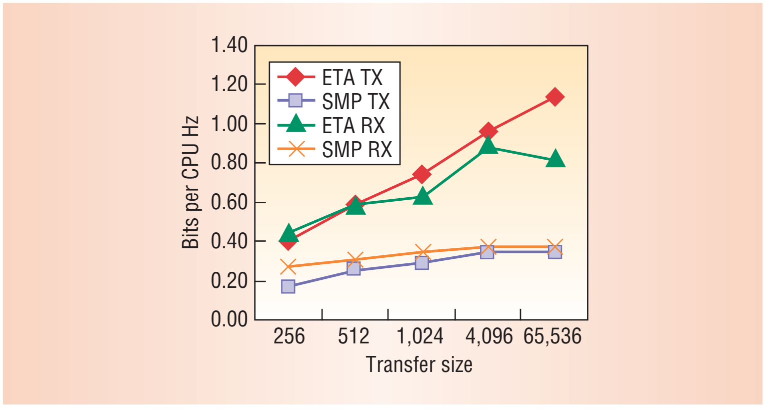


Figure 6. Processor efficiency comparison. ETA improves TCP/IP processing efficiency by reducing system overhead.

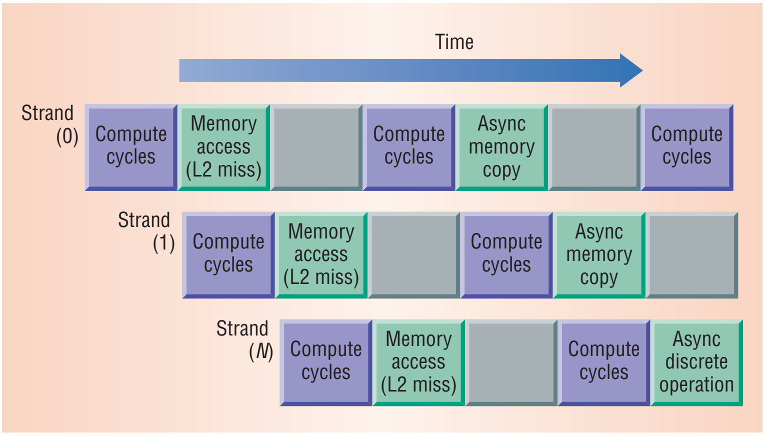


Figure 7. Lightweight threading. LWT and connection-level parallelism allow for compute-memory overlap to improve processing throughput and reduce CPU cycles.

We have recently developed a user-level version of the kernel DTI and continue to see significant efficiency improvement in user-level ETA compared to Linux and sockets. We are currently testing real applications on ETA using the direct user socket interface.[13] Although tuning is not complete, user-level ETA tests show that throughput equals the kernel level throughput, with some cases showing increased CPU utilization. Profiling to date shows that much of the additional overhead stems from our implementation of signals for synchronization.

Initial results from the ETA prototype show that partitioning the packet processing engine can significantly increase a standard multiprocessor server's overall communication performance. Specifically, partitioning the packet processing onto a dedicated set of compute resources allows optimizations that aren't possible when time-sharing the same compute resources with the OS and applications.

For example, our prototype PPE does not incur interrupt and system call overhead because it can poll shared memory for new work. It can perform this polling without placing load on the memory subsystem or the platform's front-side bus because the cache coherence protocols let the PPE poll internally in cache. The PPE only incurs memory load when either a network device or one of the host processors updates the associated memory location. Cache interference is also largely avoided because the PPE doesn't need to switch context or share caches with the OS and applications except through the ETA host interface.

Other possible optimizations include strategic prefetching of control and packet header information.

Addressing memory challenges

To address the memory challenges and copy overhead, we developed a memory-aware reference stack (MARS) for TCP/IP processing. In developing MARS, we looked for innovative mechanisms to either bring data closer to the CPU before it is accessed or to overlap the memory access latency by performing other useful computation—processing of other packets or bookkeeping operations.

In addition to the software prefetching already available on today's CPUs, MARS takes advantage of three new latency reduction techniques: lightweight threading, direct cache access, and asynchronous memory copies.

Lightweight threading. To tolerate long-latency events such as individual memory accesses and data copies, MARS employs multiple lightweight threads that execute within a single OS thread context. To differentiate them from the OS thread, we refer to these lightweight threads as strands. Strands either process independent packets (belonging to different connections) or perform bookkeeping operations.

When a strand incurs a long-latency event such as a cache miss, it switches to another strand to overlap the latency with useful computation that can be performed on another packet. For this to be effective, the strand-switching overhead must be extremely small—a fraction of the memory access latency. Therefore, in our MARS prototype, we have kept the strand context that must be saved and restored to a minimum.

Figure 7 shows how LWT exploits connection-level parallelism by overlapping packet processing with memory accesses to achieve improved processing throughput and reduce CPU utilization. In terms of implementation, detection of the memory access—a cache miss or a copy operation—and switching to the other strand can be either implicit (requiring hardware support[14,15]) or explicit (in software, as in our prototype).

Direct cache access. DCA promotes the push model of data transfer between the NIC and the CPU. Conventional platforms place incoming network data—descriptors, headers, and payload—in memory before notifying the CPU that the data has arrived. As a result, when the CPU accesses this data, it suffers cache misses and associated memory read latencies.

As Figure 8 shows, DCA helps reduce CPU read latency as well as memory bandwidth by making the incoming network data directly available in the CPU's cache. Prior to notifying the CPU of a packet's presence, MARS employs DCA to route descriptors, headers, and even payload traffic from the NIC directly to the CPU's cache.

Asynchronous memory copies. Copies can be the most time-consuming operations during TCP/IP processing. Asynchronous memory copies are hardware mechanisms that permit copies to take place asynchronously with respect to the CPU.

Figure 9 shows the typical execution flow when a hardware data mover engine is used for AMC. With this engine, once the copy operation is scheduled on this hardware, the CPU can use LWT to begin processing another packet by switching to another strand.

The considerations for copy engine implementation include the setup cost to initiate the copy and the cost to notify completion. As a result, mechanisms that keep setup and notification costs low are highly desirable. To this end, integrating the data mover engine into the CMP processor can both help reduce these costs and let the copy engine perform faster by taking advantage of the last-level cache.

With the exception of AMC, most of the mechanisms that MARS employs can be applied to multiple data structures that the TCP/IP stack touches. However, as Table 1 shows, our experimentation with the MARS prototype leads us to conclude that certain techniques yield optimal performance for certain data structures.

Further, using these techniques efficiently required revisiting the TCP/IP stack implementa-

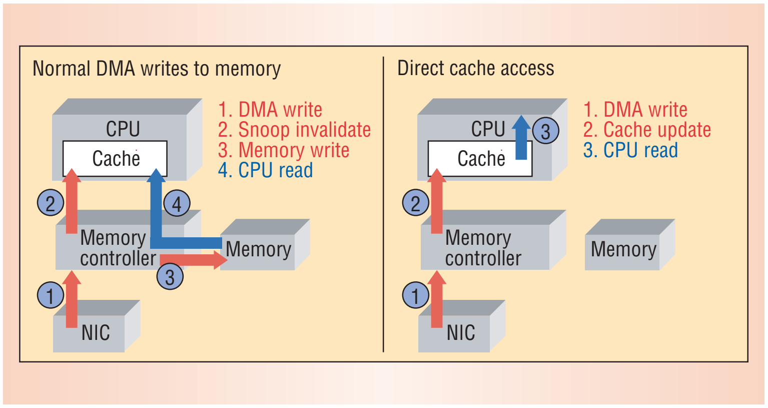


Figure 8. Direct cache access. DCA helps reduce CPU read latency as well as memory bandwidth by making the incoming network data directly available in the CPU's cache.

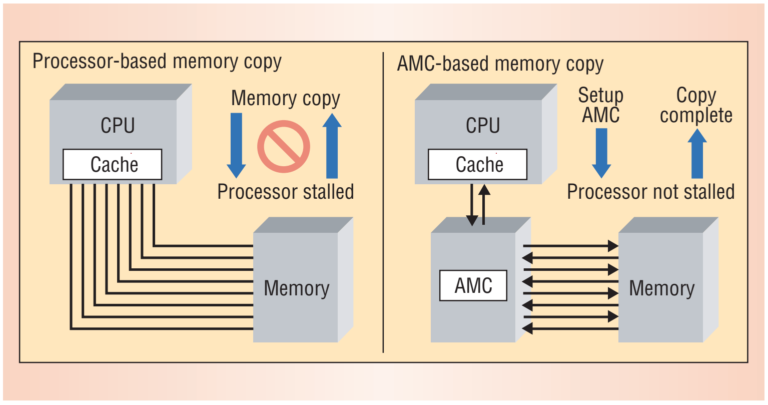


Figure 9. Asynchronous memory copies. Once the copy operation is scheduled on this hardware, the CPU can begin processing another packet and hide the memory copy latency.

Table 1. Applying latency hiding techniques.

Type of TCP/IP memory access	Software prefetching	Lightweight threading	Direct cache access	Asynchronous memory copy
Descriptors	Yes	Yes	Preferred	
Headers	Yes	Yes	Preferred	
TCP/IP control block	Yes	Preferred		
Payload	Yes		Yes	Preferred

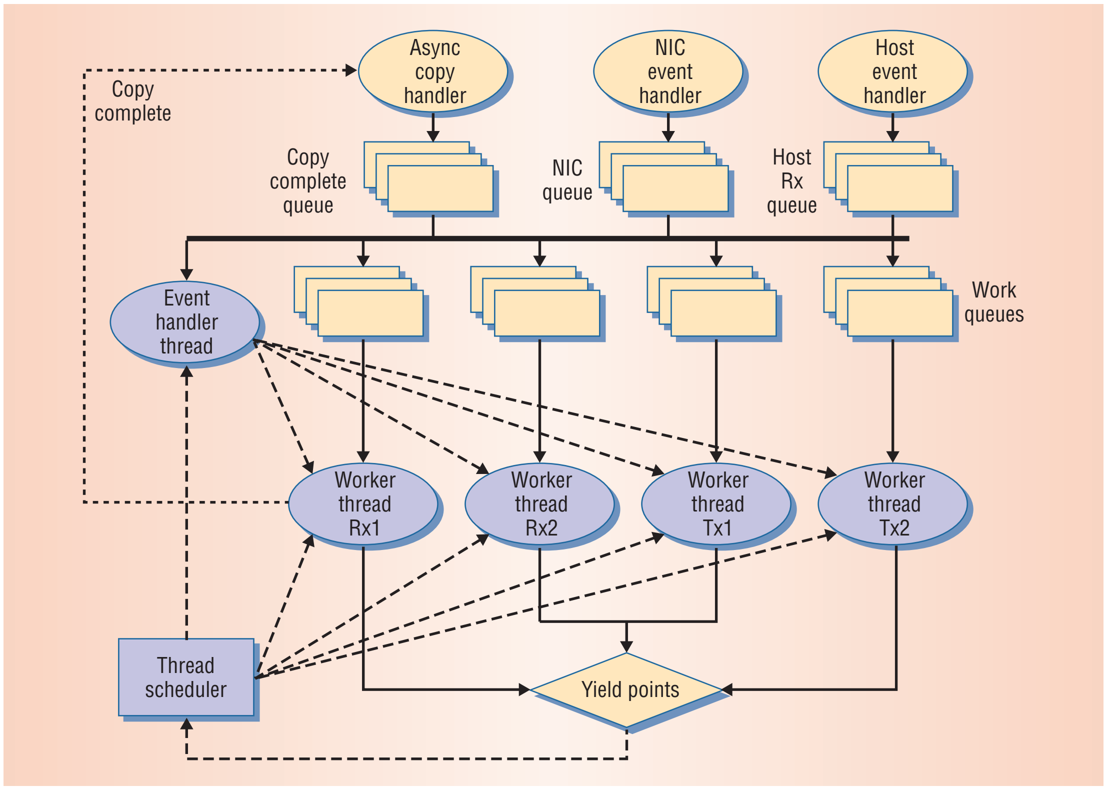


Figure 10. MARS architecture. Employing two strands each for receive and transmit operations and one strand for operations, such as event handling, yields efficient TCP/IP processing.

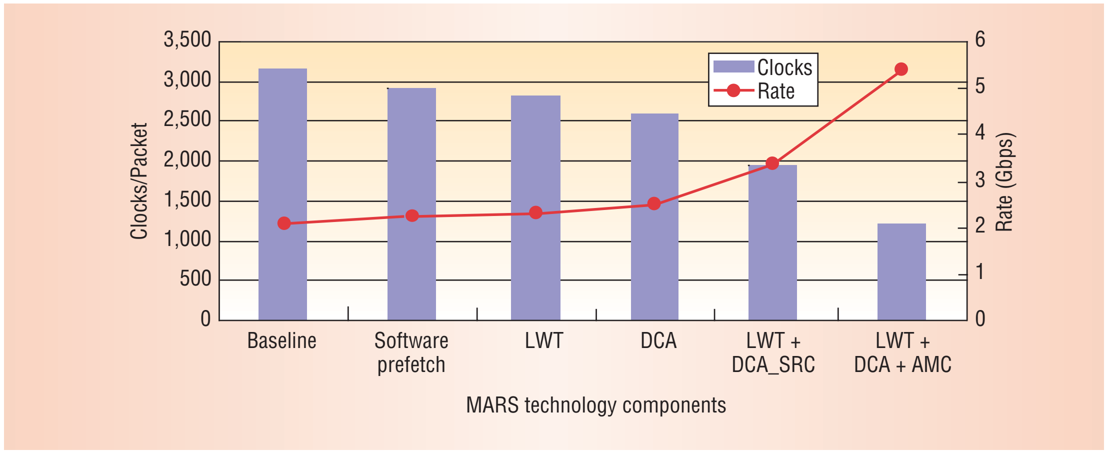


Figure 11. MARS performance summary. Combining the latency reduction techniques with the improved stack implementation achieves more than a twofold improvement.

tion. Our MARS prototype employs the LWT architecture shown in Figure 10. While this architecture is configurable, we have found that employing two strands each for receive and transmit operations and one strand for event handling and other operations, such as timers, yields efficient TCP/IP processing.

Our MARS prototype confirms the benefits of combining these latency reduction techniques with the improved stack implementation. Figure 11 shows the benefits of applying the MARS latency hiding techniques incrementally for receive-side processing of a 512-byte payload. Although each technique can be employed independently, we have found that applying them together achieves more than a twofold improvement. These benefits improve further with an increase in payload sizes. For example, for a 1,460-byte packet, the MARS prototype shows that addressing the memory stalls can result in a greater than threefold increase in achievable network bandwidth—from ~3 Gbps to ~11 Gbps.

While research in TCP/IP processing has been under way for several decades, the increasing networking needs of server workloads and evolving server architectures point to the need to explore TCP/IP acceleration opportunities.

At Intel Labs, we are experimenting with mechanisms that address system and memory stall time overheads. In our ETA project, we are currently working on enabling user applications over ETA through the sockets interface. In our MARS project, we are finalizing our reference implementation, which we plan to integrate into an OS environment to investigate the achievable gains using real applications. In addition to ETA and MARS, we are also studying the effects of interrupt- and connection-level affinity on TCP/IP processing performance.[16]

We are also beginning to explore mechanisms to support latency-critical TCP/IP usage models such as storage over IP[17] and clustered systems. The goal is to identify the right level of hardware support required for efficient and fast communication on future CMP processors and server platforms. More details on our research in this area are available at www.intel.com/labs/perfnet/. ■

Acknowledgment

We thank all the members of our team who were associated with the ETA and MARS proof-of-concept, measurement, and analysis. Without their help, this work would not have been possible.

References

1. S. Makineni and R. Iyer, "Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads," *Proc. 6th IEEE Workshop on Workload Characterization*, IEEE Press, 2003, pp. 33-41.
2. K. Kant, "TCP Offload Performance for Front-End Servers," *Proc. IEEE Global Telecommunications Conference* (GLOBECOM 03), IEEE Press, 2003, pp. 3242-3247.
3. V. Jacobson et al., "TCP Extension for High Performance," RFC 1323, LBL, ISI, and Cray Research, May 1992.
4. D. Clark et al., "An Analysis of TCP Processing Overhead," *IEEE Communications*, June 1989; www.comsoc.org/livepubs/ci1/public/anniv/clark.html.
5. J. Salim, "Beyond SoftNet," *Proc. 5th Ann. Linux Showcase and Conf.*; www.linuxshowcase.org/2001/full_papers/jamal/jamal.pdf.[27]
6. J. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *ACM Trans. Computer Systems*, Aug. 1997, pp. 217-252.
7. E. Nahum et al., "Cache Behavior of Network Protocols," *Proc. ACM Sigmetrics 97*, ACM Press, 1997, pp. 169-180.
8. W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, Mar. 1995, pp. 20-24.
9. M.N. Thadani and Y.A. Khalidi, "An Efficient Zero-Copy I/O Framework for UNIX," tech. report SMLI TR-95-39, Sun Microsystems Laboratories, May 1995.
10. J. Mogul, "TCP Offload Is a Dumb Idea Whose Time Has Come," *Proc. 9th Workshop on Hot Topics in Operating Systems* (HotOS IX), Usenix Assoc., 2003; www.usenix.org/events/hotos03/tech/full_papers/mogul/mogul.pdf.
11. G. Regnier et al., "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," *IEEE Micro*, Jan./Feb. 2004, pp. 24-31.
12. D. Dunning et al., "The Virtual Interface Architecture," *IEEE Micro*, vol. 18, no. 2, 1998, pp. 66-76.
13. Y. Turner et al., "Scalable Networking for Next-Generation Server Platforms," to be published in *Proc. 3rd Ann. Workshop on System Area Networks* (SAN-3), Madrid, 2004.
14. M. Horowitz et al., "Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications," *ACM Trans. Computer Systems*, May 1998, pp. 170-205.
15. X. Qiu and M. Dubois, "Tolerating Late Memory Traps in Dynamically Scheduled Processors," *IEEE Trans. Computers*, June 2004, pp. 732-743.
16. A. Foong, J. Fung, and D. Newell, "An In-Depth Analysis of the Impact of Processor Affinity on Network Performance," to appear in *Proc. IEEE Int'l Conf. Networks*, IEEE Press, 2004.
17. P. Sarkar et al., "Storage over IP: Does Hardware Support Help?" *Proc. 2nd Usenix Conf. File and Storage Technologies*, Usenix Assoc., 2003, pp. 231-244.

***Greg Regnier**, a member of the team that developed the virtual interface architecture, is a principal engineer in the Communications Technology Lab at Intel. His research focuses on using standard networking media and protocols to improve data center scalability and performance. He received a BS in computer science from St. Cloud State University in Minnesota. Contact him at greg.j.regnier@intel.com.*

***Srihari Makineni** is a senior engineer in the Communications Technology Lab at Intel. His research focuses on developing techniques for accelerating packet-processing applications on Intel's server platforms. Makineni received an MS in electrical and computer engineering from Lamar University, Texas. Contact him at srihari.makineni@intel.com.*

***Ramesh Illikkal** is a senior researcher at Intel. His research focuses on using modeling and simulation techniques to investigate future processor and platform architecture optimizations for better network* *performance. Illikkal received an MS in electronics and a postgraduate diploma in operations research and computer applications from Cochin University of Science and Technology. Contact him at ramesh.g.illikkal@intel.com.*

***Ravi Iyer** is a senior research scientist in the Communications Technology Lab at Intel. His research interests include computer architecture, networking protocols and acceleration, commercial workload characterization, distributed computing, and performance evaluation. Iyer received a PhD in computer science from Texas A&M University. Contact him at ravishankar.iyer@intel.com.*

***Dave Minturn** is a senior researcher in the Communications Technology Lab at Intel. His research focuses on investigating threading models and other optimizations to efficiently scale TCP/IP packet processing. Minturn received a BS in computer science from Northern Arizona University. Contact him at dave.b.minturn@intel.com.*

***Ram Huggahalli** is a senior research scientist in the Communications Technology Lab at Intel. His research focuses on developing system-level protocols that achieve faster effective data paths between CPU and I/O devices. Huggahalli received an MS in electrical engineering and in engineering management from the University of Missouri-Rolla. Contact him at ram.huggahalli@intel.com.*

***Don Newell** is a principal engineer in the Communications Technology Lab at Intel. His research interests include server architecture, networking, and I/O acceleration. Newell received a BS in computer science from the University of Oregon. Contact him at donald.newell@intel.com.*

***Linda Cline** is a senior network software engineer in the Communications Technology Lab at Intel. Her research interests include server architecture and I/O performance, network protocols and services, and networked multimedia. Cline received a BS in com- puter science from Portland State University. Contact her at linda.s.cline@intel.com.*

***Annie Foong** is a senior research engineer with Intel Labs. Her research interests include new architectures and systems software paradigms for improving server I/O performance. Foong received a PhD in electrical and computer engineering from the University of Wisconsin-Madison. Contact her at annie.foong@intel.com.*