TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance

Murali Rangarajan, Aniruddha Bohra, Kalpana Banerjee,
Enrique V. Carrera, Ricardo Bianchini
Department of Computer Science
Rutgers University, Piscataway, NJ 08854-8019
{*muralir, bohra, kalpanab, vinicio, ricardob*}*@cs.rutgers.edu*

Liviu Iftode
Department of Computer Science
University of Maryland, College Park, MD 20742
iftode@cs.umd.edu

Willy Zwaenepoel
Department of Computer Science
Rice University, Houston, TX 77005
willy@cs.rice.edu

Abstract

TCP Server is a system architecture aiming to offload network processing from the host(s) running an Internet server. The TCP Server can be executed on a dedicated processor, node, or intelligent network interface using low-overhead, non-intrusive communication between it and the host(s) running the server application.

In this paper, we present and evaluate two implementations of the TCP Server architecture: (1) using dedicated network processors on a symmetric multiprocessor (SMP) server and (2) using dedicated nodes on a cluster-based server built around a memory-mapped communication interconnect.

We have quantified the impact of offloading on the performance of network servers for these two TCP Server implementations, using server applications with realistic workloads. We were able to achieve performance gains of up to 30% with our SMP-based as well as cluster-based implementations for the scenarios we studied. Based on our experience and results, we conclude that offloading the network processing from the host processor using a TCP Server *architecture is beneficial to server performance when the server is overloaded. A complete offloading of the TCP/IP processing requires substantial computing resources on the TCP server. Depending on the application workload, either the host processor or the TCP server can become the bottleneck, stressing the need for an adaptive scheme to balance the load between the host and the TCP server.*

1 Introduction

With increasing processing power, the two main performance bottlenecks in web servers are the storage and network subsystems. A significant reduction of the impact of disk I/O on performance is possible by caching, combined with server clustering and request distribution techniques like LARD [25]. This results in removing disk accesses from the critical path of request processing. However, the same is not true for the network subsystem, where every outgoing data byte has to go through the same processing path in the protocol stack down to the network device.

In a traditional system architecture, performance improvements in network processing can come only from optimizations in the protocol processing path [1, 12, 15, 21]. As a result, increasing service demands on today's network servers can no longer be satisfied by conventional TCP/IP protocol processing without significant performance or scalability degradation. Factoring out disk I/O through caching, TCP/IP protocol processing can become the dominant overhead compared to application processing and other system overheads [20, 33]. Furthermore, with gigabit-per-second networking technologies, protocol and network interrupt processing overheads can quickly saturate the host processor at high loads, thus limiting the potential gain in network bandwidth [4].

Two solutions have been recently proposed to alleviate the overheads involved in TCP/IP networking: *(i)* offloading some (or all) of the TCP/IP processing

to intelligent network interface cards (I-NIC) capable of speeding up the common path of the protocol [3, 9, 10, 14, 17, 32] and *(ii)* replacing the expensive TCP/IP processing with a more efficient transport protocol [11], over a System Area Network (SAN), based on user-level memory-mapped communication such as VIA [13] and Infiniband [16]. The first approach alleviates the overheads associated with conventional host-based network processing, while the second one exploits the benefits of the SAN for intra-server communication. Other work has been done on confining execution of the TCP/IP protocol, system calls, and network interrupts to a dedicated processor of a multiprocessor server, but limited results have been reported [24].

We propose a generic architecture called *TCP Server*, that offloads TCP/IP processing from the server host to a dedicated processor/node. We call the dedicated processor/node which executes the TCP/IP processing, a TCP server. The performance of the *TCP Server* solution depends on two factors: *(i)* the efficiency of the TCP server implementation itself, and *(ii)* the efficiency of the communication between the host and the TCP server. The latter means that TCP/IP offloading must be implemented with low-overhead, non-intrusive communication.

In this paper, we present and evaluate two implementations of the *TCP Server* architecture. The first implementation uses one or more dedicated processors to perform TCP/IP processing in a Symmetric Multiprocessor (SMP) server. In this case, the non-intrusive communication between the host and the dedicated processor(s) is achieved using shared memory, with minimal overhead. We evaluate the server performance as a function of the number of processors dedicated for network processing and the amount of processing offloaded to them. We also study the tradeoffs between polling and interrupts for event notification in this environment.

In our second implementation, we offload the network processing to dedicated node(s) in a cluster-based server. In this case, the host and the TCP server communicate using memory-mapped communication over a high-speed interconnect [13, 16]. We investigate the design space of *TCP Servers* for clusters with memory-mapped communication using a user-level implementation over VIA and two socket interfaces: a conventional socket interface and our MemNet API [28]. The Memory-Mapped Networking (MemNet) API is a memory-mapped socket interface between the application and TCP server with support for zero-copy asynchronous communication. MemNet's use of memory-mapped communication for TCP server access is analogous to the way this technique is exploited by the Direct Access File System (DAFS) standard for remote file server access [18].

This paper represents the first study to evaluate the benefits of offloading TCP/IP processing in a comprehensive manner. We have quantified the impact of offloading on the performance of network servers for two *TCP Server* implementations, using server applications with realistic workloads. We were able to achieve performance gains of up to 30% with our SMP-based as well as cluster-based implementations for the scenarios we studied. Based on our experience and results, we conclude that offloading the network processing from the host processor using a *TCP Server* architecture is beneficial to server performance when the server is overloaded. A complete offloading of the TCP/IP processing requires substantial computing resources on the TCP server. Depending on the application workload, either the host processor or the TCP server can become the bottleneck stressing the need for an adaptive scheme to balance the load between the host and the TCP server.

The remainder of this paper is organized as follows. Section 2 describes our motivation for this work in detail. Section 3 provides an overview of the *TCP Server* architecture. Sections 4, 5 and 6 describe the details of each architecture and evaluate them. Section 7 presents the related work. Finally, Section 8 presents our conclusions.

2 Motivation

In traditional network servers, the TCP/IP protocol processing often dominates the cost incurred from application processing and other system overheads. Under heavy load conditions, network servers suffer from host CPU saturation as a result of protocol processing and frequent interruptions from asynchronous network events. In this section, we briefly explain the TCP/IP protocol processing path for various socket operations and present experimental results in support of the above statements, suggesting a need to offload networking functionality from a host.

2.1 Conventional TCP/IP Processing

In what follows, we describe conventional network processing using the Linux TCP/IP implementation. First, we describe the processing involved for the `send` and `recv` system calls on a stream socket in the TCP/IP stack.

- **send processing:** When the application performs a `send` system call, the OS copies the data to kernel buffers, to prevent it from being overwritten before being sent out, and returns to the application. This is followed by the TCP send, which makes another copy to allow retransmission of the data in case of an error. Further send processing includes dispatching of the data from the kernel buffers to the NIC.
- **receive processing:** As soon as a packet is received on the NIC, an interrupt is raised. At the end of the interrupt processing, the bottom half executes, which takes care of checksumming the packet, and demultiplexing the packet according to the protocol. This is followed by IP receive processing and TCP receive processing where the system demultiplexes the data for the destination socket and queues the received data into the receive buffers of the socket. The data is finally copied into the application buffers when the application posts a receive.

We identify five distinct components of TCP/IP processing below:

- **C1 - interrupt processing:** interrupt processing includes the time to service NIC interrupts and setup DMA transfers.
- **C2 - receive bottom:** receive processing excluding the copy into the application buffer at the time of the `recv` system call.
- **C3 - send bottom:** send processing done *after* copying the data to kernel buffers.
- **C4 - receive upper:** receive processing which copies the data into the application buffers.
- **C5 - send upper:** send processing which copies the data from the application buffers to the socket buffers inside the kernel.

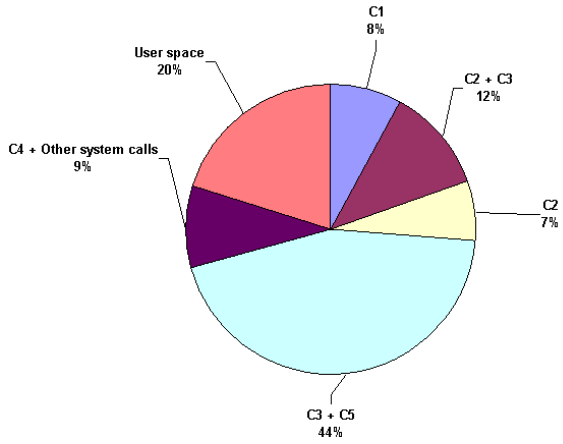

Figure 1. Apache Execution Time Breakdown

2.2 TCP/IP Overhead

We have quantified the time alloted to network processing from the execution time of an Apache (apache-1.3.20) web server. In this experiment, we used a synthetic workload of repeated requests for a 16 KB file cached in memory. Figure 1 shows the execution time breakdown on a dual Pentium 300MHz system with 512 MB RAM and 256 KB L2 cache, running Linux 2.4.16. We instrumented the Linux kernel to measure the time spent in every function inside the kernel in the execution path of `send` and `recv` system calls, as well as the time spent in interrupt processing.

The results show that the web server spends only 20% of its execution time in user space. **C1** accounts for 8% of the time. The portion of **C2** immediately after the interrupt processing combined with the portion of **C3** which involves dispatching of data from the kernel buffers to the NIC take up 12% of the time. The remainder of **C2** takes up 7% of the time. **C5** and the remainder of **C3** account for 44% of the time. **C4** is a hidden cost accounted with other system calls (9%). Altogether, network processing takes about 71% of the total execution time.

In addition to the direct effect of "stealing" processor cycles from the application, network processing also affects the server performance indirectly. Asynchronous interrupt processing and frequent context switching contribute to the overheads due to effects like cache and TLB pollution.

We conclude that offloading TCP/IP processing from the host processor to a dedicated processor can improve server performance in two ways: *(i)* by freeing up host processor cycles for the application, and *(ii)* by eliminating the harmful effects of *OS intru-*

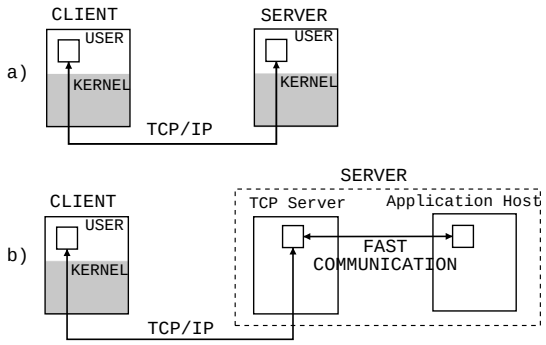

Figure 2. TCP Server Architecture

sion [23] on the application execution.

Although other operating systems have optimized TCP/IP protocol processing, we chose Linux for practical reasons: *(i)* it is open source, which makes it easier to implement the TCP Server architecture, and *(ii)* it has drivers to support memory-mapped communication over VIA. However, given that TCP/IP processing remains the major overhead for a network server under any operating system, our conclusions on TCP Server are fairly general although the performance benefits of various TCP Server optimizations may be different for different operating systems.

3 TCP Server Architecture

TCP Server is a system architecture for offloading network processing from the application hosts to dedicated processors, nodes, or intelligent devices. This separation improves server performance by isolating the application from OS networking and by removing the harmful effect of co-habitation of various OS services. Figure 2 presents two architectures for network servers: *(a)* a conventional server architecture and *(b)* an architecture based on TCP Servers. In the conventional architecture, TCP/IP processing is done in the OS kernel of the node which executes the server application. In the TCP Server architecture, the application host avoids TCP processing by tunneling the socket I/O calls to the TCP server using fast communication channels. In effect, TCP tunneling transforms socket calls into lightweight remote procedure calls.

A TCP server can execute the entire TCP processing or it can split the TCP/IP processing with the application hosts. From our discussion in Section 2, it is possible to offload components **C1**, **C2** and **C3** to the TCP server. We do not focus on offloading **C4** since we expect the benefits to be insignificant for network servers in which the receive data volume is much lower than the send volume. Offloading **C5** requires modifications to the socket API and can be achieved using the optimizations discussed below.

The performance of the TCP Server solution depends on two factors: *(i)* the efficiency of the TCP server implementation, and *(ii)* the efficiency of the communication between the host and the TCP server. As the goal of TCP/IP offloading is to reduce network processing overhead at the host, using a faster and lighter communication channel for tunneling is essential. Server performance using the TCP Server architecture can be additionally improved by optimizations that improve *(i)* and *(ii)*. In what follows, we briefly discuss these optimizations, their impact on the application programming interface and performance.

The first set of optimizations target the efficiency of the TCP server implementation.

- **S1: Avoiding interrupts.** Since the TCP server performs only the TCP/IP processing, interrupts can be easily and beneficially replaced with polling. However, the frequency of polling must be carefully controlled, as a very high rate would lead to bus congestion and a very low rate would result in inability to handle all events. The problem is aggravated by the higher layers in the TCP stack having unpredictable turnaround times and by multiple network interfaces.
- **S2: Processing ahead.** Idle cycles at the TCP Server can be used to perform certain operations ahead of time (before they are actually requested by the application). The operations that can be eagerly performed are the `accept` and `receive` system calls.
- **S3: Eliminating buffering at the TCP server.** The TCP server buffers data received from the application before sending it out to the network interface. It is possible to eliminate this extra buffering by having the TCP server send data out directly from the buffers used for communication with the application host.

Next, we present the optimizations to improve the efficiency of the interaction between the host and the TCP server.

- **H1: Bypassing the host kernel.** To achieve good performance, the application should communicate with the TCP server from user-space directly,

without involving the host OS kernel in the common case. This can be done without sacrificing protection by establishing a direct *socket channel* between the application and the TCP server for each open socket. This is a one-time operation performed when the socket is created, hence the socket call remains a system call in order to guarantee protected communication.

- **H2: Asynchronous socket API.** By using asynchronous socket calls, the application can exploit the TCP Server architecture to avoid the cost of blocking and rescheduling. Using the asynchronous API allows the application to hide the latency of a socket operation by overlapping it with useful computation.
- **H3: Avoiding data copies at the host.** To achieve this, the application must tolerate the wait for end-to-end completion of the send, i.e., when the data has been successfully received at the destination. If this is acceptable, the TCP server can completely avoid data copying on a send operation. For retransmission, the TCP server may have to read the data again from the application send buffer using non-intrusive communication. Pinning application buffers to physical memory may be necessary in order to implement this optimization.
- **H4: Dynamic load balancing.** Depending on the application workload, either the TCP server or the application host can get saturated. An adaptive scheme to balance the load or resource allocation between the application and TCP server will help server performance.

4 TCP Server Implementations

In this section, we present two implementations of the TCP Server architecture:

- On a processor dedicated to TCP/IP processing in a *symmetric multiprocessor (SMP) server*.
- On a node dedicated to TCP/IP processing in a *cluster-based server*.

Common to both implementations is a fast, low-overhead memory-mapped communication architecture between the application host and the TCP Server. The second implementation can have multiple incarnations, ranging from a front-end computer in a VIA-based multi-tier network server, to an intelligent network interface connected to an Infiniband-based server.

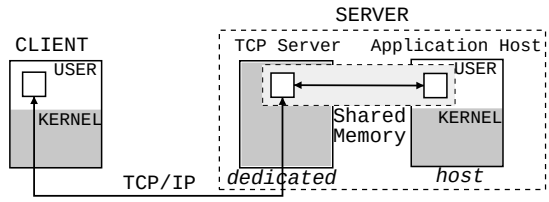

Figure 3. TCP Server in an SMP-based server

Implementation	C1	C2	C3	H1	H2	H3	H4	S1	S2	S3
SMP_base										
SMP_C1C2	x	x								
SMP_C1C2S1	x	x						x		
SMP_C1C2C3	x	x	x							
SMP_C1C2C3S1	x	x	x					x		

Table 1. SMP-based implementations of TCP Server

4.1 TCP Server in SMP-based Servers

We partition the set of processors in an SMP-based server into *host* and *dedicated* processors. The *dedicated* processors are used exclusively by the TCP server for TCP/IP processing. The communication between the application and the TCP server is through queues in shared memory as shown in Figure 3.

4.1.1 System Overview

Network generated interrupts are routed exclusively to the *dedicated* processors. The TCP server executes a tight loop in the kernel context on each dedicated processor. On a socket send, the data to be sent is copied from the application to a kernel buffer. This buffer is part of the shared memory queue, from where the TCP server dequeues the offloading request and performs **C3**. The TCP server finally sets up a DMA to the NIC.

The receive events, which are asynchronous, are routed to the TCP server, which performs **C1** and **C2**. On a receive call from the application, **C4** is performed on the host processor.

In Table 1, we present the different implementations, the functionality mapped to the TCP server and the optimizations used in each of these implementations. SMP_base refers to the unmodified Linux TCP/IP implementation on the SMP system. In each of the other implementations, the interrupts and the bottom half processing are executed on the TCP server.

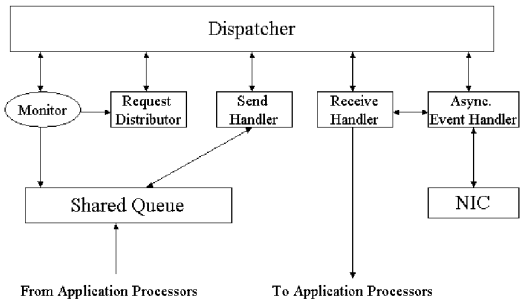

Figure 4. Organization of the TCP Server in an SMP-based server showing the different modules and their interaction with each other

Our architecture, where network processing is limited to the dedicated processors, allows us to poll on the network interface frequently without slowing other tasks down. We study polling in the dedicated processor as an alternative way to handle the events at the network interface.

4.1.2 Implementation Details

We modified the Linux-2.4.16 kernel to implement the TCP Server architecture on an SMP-based server. The TCP server executes in the kernel context on each dedicated processor where no application processing takes place. This is done by executing a kernel thread which never yields to the user level programs. Moreover, all interrupts, except for the system clock interrupt, are routed away from the dedicated processors using the external IO/APIC routing mechanism. The kernel thread runs the TCP server *dispatcher* which schedules the other components of the system. The organization of an SMP-based TCP server is shown in Figure 4. The different components of a TCP server are: *(i)* the request distributor, *(ii)* the queue monitor, *(iii)* the send request handler, *(iv)* the receive request handler, *(v)* the asynchronous event handler, and *(vi)* the shared queue.

The shared queue is a circular queue of send requests with references to the process, socket, and the data buffers associated with each request. We can also assign priorities to the send requests to ensure an order in which the requests are serviced. The default policy is FIFO.

The dispatcher schedules the request distributor periodically. It can also be scheduled after an asynchronous event or on a trigger from the monitor. The request distributor checks the shared queue and signals the presence of a request to the dispatcher, which then calls the send request handler to carry out the necessary processing.

The receive request handler is executed by the dispatcher upon a receive event, signalled to it by the asynchronous event handler. The interaction between the TCP server and the network interface card is handled by the asynchronous event handler. The asynchronous event handler can be implemented as an interrupt service routine or as a polling handler. In the first case, it is automatically called in case of an interrupt. For the second case, the dispatcher must execute the handler so that networking events are not missed in case of delayed processing. We use a Soft-Timers [6] like mechanism by using the clock interrupt handler to guarantee the execution of the asynchronous event handler at every clock interrupt.

Each component above is added as a loadable kernel module. Some modules such as the send request handler are optional, as in the case of SMP_C1C2 and SMP_C1C2S1, no send processing is carried out in the dedicated processor, making the module unnecessary. The minimal set of modules required for the TCP server execution are the dispatcher and the asynchronous event handler.

To identify the existing modules in the system and to notify the dispatcher of an event pending for a component, we use a mechanism similar to the Linux software interrupt handlers. There is a list of registered components, along with the handler for each of them. This provides a mechanism to dynamically add or remove components of the TCP server.

The SMP-based implementation includes support to dynamically increase or reduce the size of the processor set allocated to the TCP server. At system initialization, the kernel threads are started on all available processors in the system. A subset of these sleep, allowing application processing to take place. In case of high volume of send requests, the monitor can wake up these threads, which then execute the TCP server till the monitor signals them to sleep again. For the purpose of load balancing and book-keeping needed for dynamic reconfiguration of the system, the dispatcher periodically calls the monitor, which triggers a reconfiguration based on the current state of the shared queue, the load on the system or any other policy. The default policy monitors the length of the queue and maintains the low and high watermarks to trigger reconfiguration.

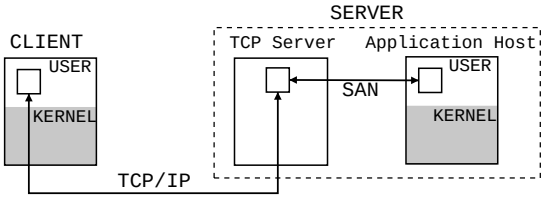

Figure 5. TCP Server in cluster-based servers

4.2 TCP Server in Cluster-based Servers

In a cluster-based server, the application host and the TCP server are PCs connected by a VIA-based SAN as shown in Figure 5. The TCP server acts as the network endpoint for the outside world. Network data is tunneled between the application host and the TCP server across the SAN using low latency memory-mapped communication.

4.2.1 System Overview

The socket call interface is implemented as a user-level communication library on the application host. The library manages and maintains VIs on the host and communicates with the TCP server using VI channels. The mapping from a socket to a VI channel is established at the time of the first operation on the socket. With this library, a socket call is tunneled through a VI channel to the TCP server. On the TCP server, a *socket provider* module interprets the socket call and performs the corresponding socket operation. The results are returned to the application host after the operation completes.

In Table 2, we present the different implementations, the functionality mapped to the TCP server and the optimizations used in each of these implementations. Cluster_base refers to the standalone host-based Linux TCP/IP implementation on the system. Since all the socket operations are offloaded to the TCP server, **C1**, **C2** and **C3** are offloaded to the TCP server by default. Implementing the cluster-based TCP Server in user-space makes it possible for us to implement optimizations **H1**, **H2** and **H3**. We also implemented optimization **S2** as follows:

- **Eager Receive** is an optimization for the network receive processing. The TCP server eagerly performs receive operations on behalf of the host and when the application issues a receive call, data is transferred from the TCP server to the application host. The TCP server posts receive for a number of bytes, and continues with further eager receive processing depending on the rate of data consumed by the host. The *socket provider* uses the `poll` system call to verify if any data is ready to be read from that socket before issuing an eager `recv`. The *socket provider* keeps the received data on the TCP server and transfers it directly into the application buffers when the application invokes a receive.
- **Eager Accept** is an optimization to the connection processing. A dedicated thread of the TCP server eagerly accepts connections upto a predetermined maximum. When the application issues an `accept`, one of the previously accepted connections is returned.

4.2.2 Implementation Details

Each socket used by the application is mapped to a VI channel and has a corresponding socket endpoint on the TCP server. The system associates a registered memory region with each VI channel which is used internally by the system. Since the mapping of a socket to a VI and its associated memory regions is maintained for the lifetime of the socket, these memory regions can be used by the system to perform RDMA transfers of control information and data between the application and the TCP server. These memory regions include the send and receive buffers associated with each socket. An RDMA-based signalling scheme is used for flow control between the application and the TCP server, for using the socket send and receive buffers.

As creating VIs and connecting them are expensive operations, the socket library on the application host creates a pool of VIs and requests connections on them from the TCP server at the time of initialization. The TCP server is implemented as a multi-threaded user-level process running on the network-dedicated node. The main thread of the TCP server accepts or rejects VI connection requests from the host depending on its existing load. On accepting a VI connection request, the main thread then hands over this VI connection to a worker thread which is then responsible for handling all data transfers on that VI.

Server applications use the MemNet API [28] to access the networking subsystem in our prototype. The MemNet API allows applications to perform sends and receives both synchronously and asynchronously. The

Implementation	C1	C2	C3	H1	H2	H3	H4	S1	S2	S3
Cluster_base										
Cluster_C1C2C3H1	x	x	x	x						
Cluster_C1C2C3H1H3	x	x	x	x		x				
Cluster_C1C2C3H1H2H3	x	x	x	x	x	x				
Cluster_C1C2C3H1H2H3S2	x	x	x	x	x	x			x	

Table 2. Cluster-based implementations of TCP Server

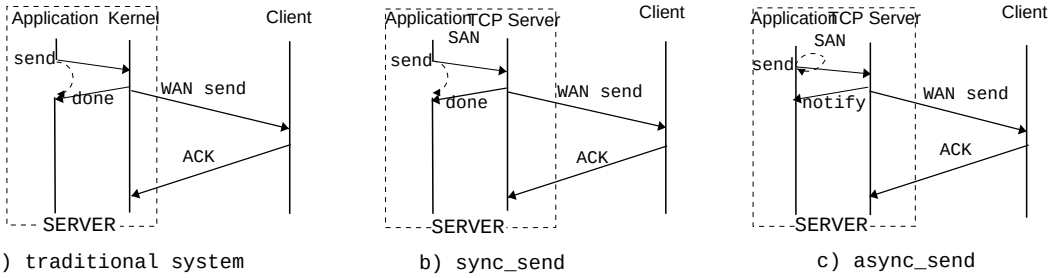

Figure 6. Comparison of the `send` operation in a traditional system with `sync_send` and async_send

send/receive primitives provided by the MemNet API allow data to be transferred directly to and from application buffers. In order to achieve this, the application needs to register its communication buffers with the system. The `register_mem` and `deregister_mem` primitives enable the application to register and deregister memory with the system.

The `sync_send/sync_recv` primitives return to the application only after the send/receive operation is offloaded to the TCP server. The `async_send/async_recv` primitives immediately return job descriptors to the application. The job descriptors can be used by the application to check the completion status of asynchronous operations. The application has the option of using the `job_wait` or `job_done` primitives to wait or poll respectively, for completion of the asynchronous operation specified in the job descriptor. To guarantee correctness, the system assumes that applications do not overwrite buffers specified as part of an asynchronous operation, before the operation completes.

The *socket provider* uses the standard Linux socket implementation in our prototype. This guarantees reliable transmission of data once a socket send is performed on the TCP server. In `sync_send`, control returns to the application only after the entire buffer is sent using the TCP/IP socket provider. In `async_send`, control returns to the application as soon as the send is posted on the VI channel corresponding to the socket. The application has to avoid overwriting buffers used in asynchronous sends until the operation completes. Figure 6 compares the `send` in a traditional system with the `sync_send` and the `async_send`. In the figure, the dotted arrow indicates the return of control to the application. The gap in the application processing indicates the duration for which the application is blocked on the `send` operation.

5 Experimental Setup

For the SMP-based implementation, we used two configurations: *(i)* a 550 MHz Intel Xeon-based 4-way SMP system with 1GB DRAM and 1MB L2 cache. *(ii)* a 300 MHz Intel Pentium-based 2-way SMP system with 512 MB DRAM and 256 KB L2 cache. Both configurations used a 3Com 996-BT gigabit Ethernet adapter. For the cluster-based TCP server implementation, we used two 300 MHz Pentium II PCs that communicate over 32-bit 33 MHz Emulex cLAN interfaces and an 8-port Emulex switch. The TCP Server was installed with a 3Com 996B-T Gigabit Ethernet adapter. All the systems ran Linux-2.4.16 kernels.

For the 4-way SMP based TCP server evaluation, we used the Apache 1.3.20 web server [5] as the server application. The requests were generated using sclients[1] [7] driven by three different traces shown in Table 3

Currently, the socket library in our cluster-based implementation does not support primitives like `select` on the socket descriptors. This prevents us from using web server applications like Apache which uses

[1]We used sclients instead of httperf in this case as httperf could not generate enough load to saturate the 4-way SMP system.

`select` extensively. We built a multithreaded web server which services http requests from clients without using `select`. We used our custom built web server to study the performance of both implementations of the TCP Server, with a uniform workload. We used the 2-way SMP configuration for the SMP-based implementation. The requests for the files were generated by a client with httperf [22] using both HTTP/1.0 and HTTP/1.1 protocols. The client used a synthetic trace, in which 16KByte files are repeatedly requested.

6 TCP Server Evaluation

In this section, we present an evaluation of the performance impact of the TCP Server architecture for both SMP-based and cluster-based servers.

6.1 Evaluation of SMP-based Implementations

In this section, we evaluate several alternative TCP Server implementations for the SMP system. We vary the number of processors dedicated to the network processing, the amount of processing offloaded to the dedicated processors, and the event notification mechanism for the system.

We study the performance of a server system for each of the above implementations, comparing them against the unmodified uniprocessor and multiprocessor kernels. We also study the effect of the number of dedicated network processors on the performance of the server system.

To study the behaviour of SMP-based TCP server implementations, we first describe the performance evaluation for the 4-way SMP system.

6.1.1 Results for 4-way SMP

We used three traces to drive our experiments: Forth, Rutgers, and Synthetic. Forth is from the FORTH Institute in Greece. Rutgers contains the accesses made to the main server at the Department of Computer Science at Rutgers University in the first 25 days of March 2000. Synthetic is a synthetic trace in which 16-KByte files are requested. Table 3 describes the main characteristics of these traces.

Throughput: In Figure 7, we show the throughput attained by the different SMP-based TCP Server implementations at saturation. For each of the ten configurations, we plot the performance using the three traces. For the sake of clarity, we present only the

Logs	# files	Avg file size	# requests	Avg req size
Forth	11931	19.3 KB	400335	8.8 KB
Rutgers	18370	27.3 KB	498646	19.0 KB
Synthetic	128	16.0 KB	500000	16.0 KB

Table 3. Main characteristics of WWW server traces

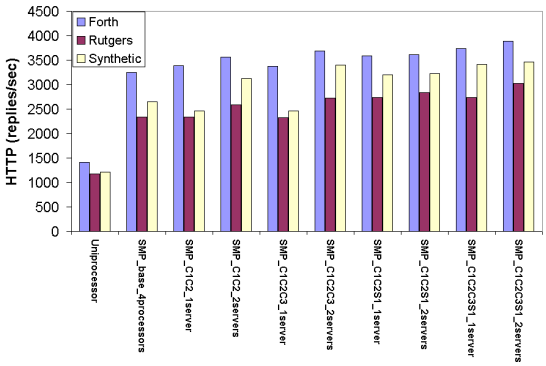

Figure 7. Throughput at saturation for Apache on a 4-Way SMP Server.

throughput at which each of the configurations saturates.

The first interesting observation we can make from this figure is that the different traces lead to similar performance trends, even though their average requested file sizes are different. Another interesting observation is that dedicating two processors to network processing is better than dedicating only one. Dedicating a processor to the TCP/IP processing is beneficial in all the cases. We also observe that dedicating more than one processors is helpful in further improving the performance of the system. Finally, configurations that use polling instead of interrupts consistently outperform the ones using interrupts. However, offloading the send processing and polling(SMP_C1C2C3S1), are more beneficial when two processors are dedicated to the network processing. Overall, we can see that offloading the network processing can achieve improvements in throughput of up to 25-30% in the cases of Rutgers and Synthetic with two dedicated processors and polling. This result demonstrates that this TCP server architecture can indeed provide significant performance gains.

CPU Utilization: Figure 8 provides more insight into these results. The figure depicts the breakdown of the average CPU utilization of the application and network processors for the different configurations we study, using the Synthetic trace. Each bar is broken into user, system, and idle times.

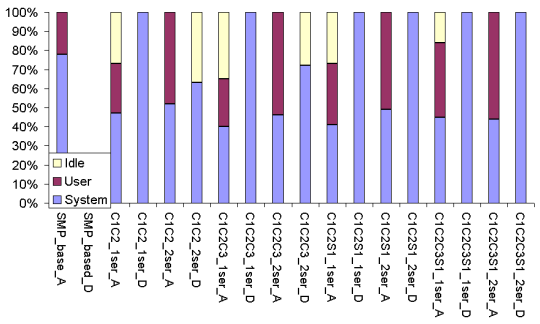

Figure 8. CPU usage at saturation for Apache on a 4-Way SMP Server. The suffix A denotes application processors and D denotes dedicated processors. The times are averaged over the entire set of processors.

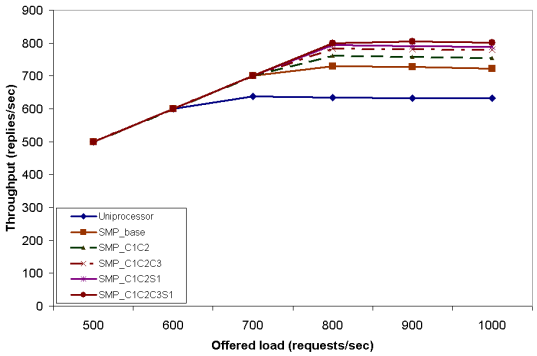

Figure 9. Throughput for a simple web server on a 2-Way SMP system using HTTP/1.0.

The figure shows that, when only one processor is dedicated to the network processing, the network processor becomes a bottleneck and, consequently, the application processor is not fully utilized and has idle time. Since the network processor is already a bottleneck, it is clear that loading it further with send operations will only degrade performance. With two network processors, there is enough processing power to handle the network processing, and the application processor becomes the bottleneck. In this case, offloading the send operations to the network processors is beneficial, as shown in the figure. (Note: Our implementations using polling (SMP_C1C2S1 and SMP_C1C2C3S1) with two network processors, do not show any idle time for the network processors. The reason is that we categorize their polling time as system time, rather than idle time). Overall, these results clearly indicate that the best system would be one in which the division of labor between the network and application processors is more flexible, allowing for some measure of load balancing. We are currently evaluating an implementation that performs such load balancing.

Our experiments reveal that using an SMP-based TCP Server implementation, the performance of a a typical web server improves by up to 28%. We observed that dedicating processors to asynchronous event handling improves the performace of a typical web server. Using polling instead of interrupts as the asynchronous event notification mechanism also improves the performance of the system. Our results also indicate that the number of dedicated processors required depends on the application workload.

6.1.2 Results for 2-way SMP

We describe the performance evaluation for the 2-way SMP system to present a uniform workload across the SMP-based and cluster-based implementations of the TCP Server. We studied the performance of our simple multithreaded web server on a 2-way SMP system running different implementations of the TCP server, described in Table 1. We varied the rate of requests and measured the rate of succesful HTTP replies as the throughput of the web server. We used both HTTP/1.0 and HTTP/1.1 protocols to measure server performance with this synthetic workload.

Throughput: Figure 9 shows the throughput for the simple web server for different kernel configurations using the HTTP/ 1.0 protocol and Figure 10 shows the throughput using HTTP/1.1 protocol. For the HTTP/1.1 protocol, we send requests for six files on every open connection in bursts of three.

In both cases, we see that offloading TCP processing to dedicated processors improves the performance of the system. In the case of HTTP/1.0, we see that the performance of the server increased by up to 10% using the TCP Server implementation. Even in the case of a more efficient protocol (HTTP/1.1), with features aimed at reducing networking overheads for application servers, we see that our system is able to provide improvement of about 6%. In both cases, the major performance benefit is due to the removal of asynchronous network events from the host processor. We can also see that the offloading of send processing helps only to a limited extent. This behaviour is due to the dedicated processor saturating before the host processor and becoming the bottleneck in the system.

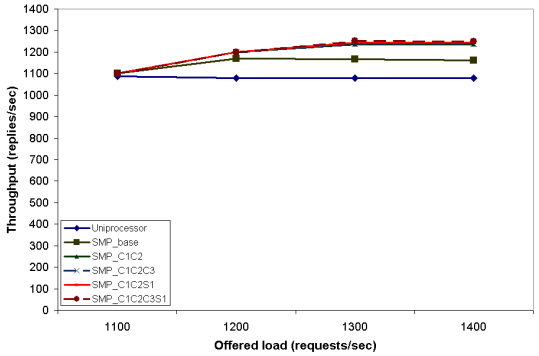

Figure 10. Throughput for a simple web server on a 2-Way SMP system using HTTP 1.1

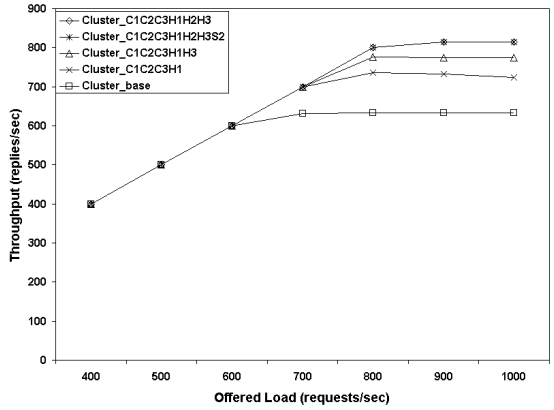

Figure 11. Throughput for a simple web server on a cluster-based TCP Server using HTTP/1.0

6.2 Evaluation of Cluster-based Implementation

We studied the performance of a simple multi-threaded web server on the four cluster-based implementations of the TCP server, described in Table 2.

6.2.1 Results

We evaluate the cluster-based TCP Server architecture by analyzing the performance of a simple multi-threaded web server. We compare the performance of the web server using the traditional socket API in our prototype (Cluster_C1C2C3H1) and using the primitives provided by the MemNet API (Cluster_C1C2C3H1H3 and Cluster_C1C2C3H1H2H3) which require buffers used in communication to be pre-registered. In Cluster_C1C2C3H1H3, the web server implementation uses the `sync_send` primitive and in Cluster_C1C2C3H1H2H3, it uses the `async_send` primitive. We also present the performance of the web server using a standalone Linux host-based socket implementation Cluster_base for comparison.

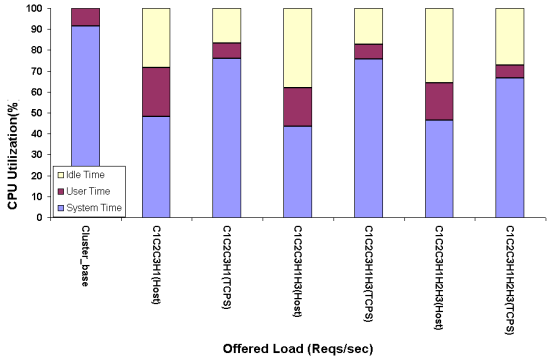

Figure 12. CPU utilization for a simple web server on a cluster-based TCP Server using HTTP/1.0

Throughput with HTTP/1.0: Figure 11 shows the throughput of the web server as a function of the offered load in requests/second. All systems are able to satisfy the offered load at low request rates. At high request rates, we see a difference in performance when Cluster_base saturates at an offered load of 700 requests/second. The web server shows an improvement of 15% in performance with Cluster_C1C2C3H1 over Cluster_base. Using the synchronous primitives (Cluster_C1C2C3H1H3), the web server is able to achieve a performance improvement of 22%. Cluster_C1C2C3H1H2H3 shows a performance gain of about 30% with the web server using asynchronous primitives like `async_send`. Cluster_C1C2C3H1H2H3 allows a better pipelining of network sends and helps the application overlap the latency of offloading the send primitive over the SAN with computation at the host. Cluster_C1C2C3H1H2H3S2, which includes the *Eager Accept* optimization, provided no additional gain since it is not the connection time, but the actual request processing time that dominates the network processing.

For optimization S2, we also observed that the *Eager Receive* optimization (not presented) does not contribute to any performance gain. In the *Eager Receive* implementation, the TCP server uses the `poll` system call to verify if data has arrived on a given socket. This leads to a slight performance degradation at high request rates by taking up some CPU time when the TCP server is already saturated.

CPU Utilization with HTTP/1.0: In Figure 12, we present the CPU utilization on the application host (Host) and TCP server (TCPS) for the four implemen-

tations, for the load at which Cluster_base saturates. At this load, the host CPU saturates for Cluster_base whereas the Cluster_C1C2C3H1H3(Host) and Cluster_C1C2C3H1H2H3(Host) have about 40% idle time. With Cluster_C1C2C3H1, since the web server uses only the traditional socket based API, it does not preregister buffers used in communication. As a result, copies take up CPU time and reduce the idle time in Cluster_C1C2C3H1(Host) to 29%. The CPU utilization on the TCP server (TCPS) shows that the TCP/IP processing overhead has been shifted to the TCP server in the offloading-based implementations. We have also observed that at higher loads, the network processing at the TCP server proves to be the bottleneck and eventually saturates the processor on the TCP server. It is interesting to note that the host processor incurs high system time overhead (about 50%) even after offloading TCP/IP processing to the TCP server. We observed that on our system, a simple ping-pong utility (tvia) provided with the VIA implementation from Emulex has a system time overhead of 30% when using 16KB packets on a single VIA connection. Taking into account the file system overhead (roughly 10%) for the web server, we can account for the system time overhead on the host processor. We are currently trying to understand the system time overhead arising from the VIA implementation to see how this can be avoided.

Throughput with HTTP/1.1: HTTP/1.1 includes features to alleviate some of the TCP/IP processing overheads. The use of persistent connections enables reuse of a TCP connection for multiple requests and amortizes the cost of connection setup and teardown over several requests. HTTP/1.1 also allows for pipelining of requests on a connection. The workload used for this study is the same as that used for HTTP/1.0. However, multiple requests (six) were sent over each socket connection, in bursts of three. Figure 13 shows the web server throughput in this case. The performance gain achieved by Cluster_C1C2C3H1H3 is about 12%, and by Cluster_C1C2C3H1H2H3 is 20%, over that of Cluster_base. These performance gains, are lower than those achieved with HTTP/1.0. However, they show that our system is able to provide substantial gains over that of a traditional networking system, even while using HTTP/1.1 features aimed at reducing networking overheads for application servers.

Greater gains are not possible with this workload because the TCP server node becomes the bottleneck at high loads. In fact, this explains why our optimizations of *Eager Receive* and *Eager Accept* (S2), do not improve throughput beyond that of Cluster_C1C2C3H1H2H3. These optimizations are intended to improve the performance of the host application at the cost of more processing at the TCP server node. However, speeding up the host does not really help overall performance because, at some point, the performance becomes limited by the TCP server node. This problem can be alleviated in three different ways: by adaptively balancing the load between the application host and TCP server, by using a faster TCP server, or by using multiple TCP servers per application node. We are currently investigating these approaches.

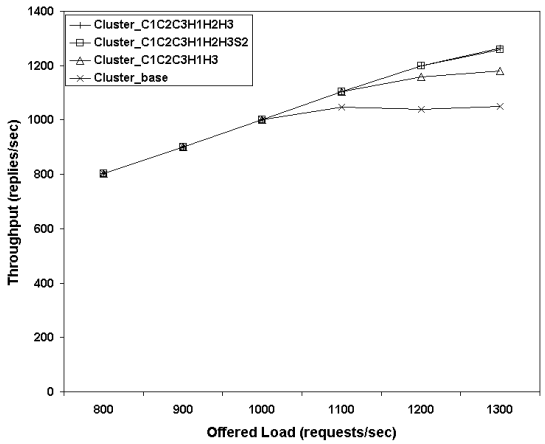

Figure 13. Throughput for a simple web server on a cluster-based TCP Server using HTTP/1.1

7 Related Work

OS mechanisms and policies specifically tailored for servers have been proposed in [8, 12, 26]. However, they do not study the effect of separating the application processing from network processing or shielding the application from OS intrusion.

An important factor in the performance of a server is its ability to handle extremely high volume of receive requests. Under such conditions, the system enters a *receive livelock* [20]. Several researchers suggest the use of polling on the system to prevent receive livelock and for high performance [6, 19, 30]. In Piglet [24], the application is isolated from the asynchronous event handling using a dedicated polling processor in an SMP.

In the Communication Services Platform (CSP) [29] project, the authors suggest a system

architecture for scalable cluster-based servers, using dedicated network nodes and a VIA-based SAN to tunnel TCP/IP packets inside the cluster. CSP was an architecture aimed to offload the network processing to dedicated nodes. However, their results are very preliminary and their goal was limited to using dedicated nodes for network processing in a multi-tier data center architecture.

Recently released network interface cards have been equipped with hardware support to offload TCP/IP processing from the host [2, 3, 10, 14, 17, 32]. Some of these cards also provide support to offload networking protocol processing for network attached storage devices including iSCSI, from software on the host processor to dedicated hardware on the adapter.

QPIP [9] is an attempt to provide a lightweight protocol for applications which offloads network processing to the Network Interface Card (NIC). However, they implement only a subset of TCP/IP on the NIC. QPIP suggests an alternative interface to the traditional sockets API but does not define a programming interface that can be exploited by applications to achieve better performance. Moreover, performance evaluation presented in [9] was limited to communication between QP-aware applications over a SAN.

Sockets Direct Protocol (SDP) [27] originally developed to support server-clustering applications over VI architecture, has been adopted as part of the InfiniBand specification. The SDP interface makes use of InfiniBand capabilities and acceleration, while emulating a standard socket interface for applications.

Voltaire has proposed a TCP Termination Architecture [31] with the goals of solving the bandwidth and CPU bottlenecks which occur when other solutions such as IP Tunneling or bridging are used to connect InfiniBand Fabrics to TCP/IP networks.

Direct Access Transport (DAT) [11] is an initiative to provide a transport exploiting remote memory capabilities of interconnect technologies [13, 16]. However, the objective of DAT is to expose the benefits of remote memory semantics only to intra-server communication.

We propose and evaluate the TCP Server architecture to offload TCP/IP processing in different scenarios for network servers. We extend this line of research and explore the separation of functionality in a system. We study the impact of separation of functionality not only for a bus-based multiprocessor system, but also for a switch-based cluster of dedicated processors.

8 Conclusions

In this paper, we introduced a network server architecture based on offloading network processing to dedicated TCP servers. We have implemented and evaluated TCP Servers in two different architectural scenarios: using a dedicated network processor in a symmetric multiprocessor (SMP) server and using a dedicated node on a cluster-based server built around a memory-mapped communication interconnect. Using our evaluations, we have quantified the impact of TCP/IP offloading on the performance of network servers.

Based on our experience and results, we draw several conclusions: (i) offloading TCP/IP processing is beneficial to overall system performance when the server is overloaded (performance gains of upto 30% were achieved in the scenarios we studied) (ii) TCP servers require substantial computing resources for complete offloading. (iii) the type of workload plays a significant role in the efficiency of TCP servers. We observed that, depending on the application workload, either the host processor or the TCP Server can become the bottleneck. Hence, a scheme to balance the load between the host and the TCP Server would be beneficial for server performance.

We are in the process of performing more experiments with each implementation, implementing dynamic load balancing between processors of different classes and implementing an optimized TCP Server for the cluster-based implementation.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, 1993.

[2] Adaptec ASA-7211 and ANA-7711. http://www.adaptec.com.

[3] Alacritech Storage and Network Acceleration. http://www.alacritech.com.

[4] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 USENIX Technical Conference*, pages 143–154, June 1998.

[5] Apache http server reference manual. http://httpd.apache.org/docs.

[6] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[7] G. Banga and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[9] P. Buonadonna and D. Culler. Queue-Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th Annual Symposium on Computer Architecture*, May 2002.

[10] Cyclone Intelligent I/O. http://www.cyclone.com.

[11] The DAT Collaborative. http://www.datcollaborative.org.

[12] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation*, pages 261–275, 1996.

[13] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2), 1998.

[14] Emulex, Inc. http://www.emulex.com.

[15] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[16] The Infiniband Trade Association. http://www.infinibandta.org, August 2000.

[17] Intel Server Adapters. http://www.intel.com.

[18] J. Katcher and S. Kleiman. An Introduction to the Direct Access File System, 6 2000.

[19] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts, and thread management. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996.

[20] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, pages 99–111, Berkeley, CA, USA, Jan. 1996.

[21] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.

[22] D. Mosberger and T. Jin. httperf – a tool for measureing web server performance, 1998.

[23] S. Muir and J. Smith. AsyMOS - An Asymetric Multiprocessor Operating System. In *Proceedings of Open Architectures and Network Programming*, San Francisco, CA, April 1998.

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

[25] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[26] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[27] J. Pinkerton. SDP: Sockets Direct Protocol. In *Infiniband Developers Conference*, Fall 2001.

[28] M. Rangarajan, K. Banerjee, and L. Iftode. MemNet: Memory-Mapped Networking for Servers. Submitted for publication, Rutgers University, Department of Computer Science Technical Report, DCS-TR-485, May 2002.

[29] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, and R. S. Madukkarumukumana. CSP: A Novel System Architecture for Scalable Internet and Communication Services. In *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[30] J. M. Smith and C. B. S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

[31] Voltaire TCP Termination Architecture. http://www.voltaire.com/pdf/Breaking through the bottleneck.pdf.

[32] Tornado for Intelligent Network Acceleration. http://www.windriver.com.

[33] Q. Y. Yiming Hu, Ashwini Nanda. Measurement analysis and performance improvement of the apache web server. Technical Report 1097-0001, University of Rhode Island, Department of Electrical and Computer Engineering, October 1997.