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

Abstract

TCP Server is a system architecture aiming to offload network processing from the host(s) running an Internet server. The basic idea is to execute the TCP/IP processing on a dedicated processor, node, or device (the TCP server) using low-overhead, non-intrusive communication between it and the host(s) running the server application.

In this paper, we propose, implement, and evaluate the TCP Server architecture to offload TCP/IP processing in two different scenarios (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 such as VIA.

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 demand substantial computing resources for complete offloading. Complete TCP/IP offloading to intelligent devices requires the device to be computationally powerful to outperform traditional architectures. (iii) the type of workload plays a significant role in the efficiency of TCP servers. 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 to server performance.

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 [28] which 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 for network processing can only come from optimizations in the protocol processing path [1, 12, 16, 23, 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. When factoring out disk I/O through caching, TCP/IP protocol processing can become the dominant overhead compared to application processing and other system overheads [22, 37]. Furthermore, with gigabit-per-second networking technologies, protocol and network interrupt processing overheads can quickly saturate the host processor with increasing network processing loads, thus limiting the potential gain in network bandwidth [4].

Two non-conventional architectures have been proposed to alleviate the overheads involved in TCP/IP networking: *(i)* offloading some of the TCP/IP processing to intelligent network interface cards (I-NIC) capable of speeding up the common path of the protocol [3, 8, 10, 14, 18, 36] and *(ii)* replacing the expensive TCP/IP processing with a lightweight, more efficient transport protocol [8, 11], using user-level and memory-to-memory communication based on standards such as VIA [13] and Infiniband [17]. The first approach attempts to alleviate the overheads associated with conventional host-based network processing and the second approach attempts to achieve better server performance within the data center by exploiting the characteristics of the underlying SAN. 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 [26].

Our work aims to understand the design, implementation, and performance of server architectures that rely on TCP/IP offloading for client-server communication. Our approach consists of decoupling the TCP/IP protocol stack processing from the server host, and executing it on a dedicated processor/node. We call this a *TCP Server* architecture. The performance of the *TCP Server* solution depends on two factors: *(i)* the efficiency of the communication between the host and the TCP server and *(ii)* the network programming interface provided to the server application. With respect to the communication efficiency, TCP servers can dramatically benefit from using low-overhead, non-intrusive, memory-mapped communication and new I/O switch technologies [17, 32]. With respect to the network API, to fully exploit the performance potential of the TCP-server and avoid data copying, the server application must use and tolerate asynchronous socket communication. In MemNet [31], we introduced the Memory-Mapped Networking API which enables applications to offload the TCP/IP processing to TCP Servers efficiently over memory-mapped interconnects. A similar technique is used by the Direct Access File System (DAFS) standard which exploits memory-mapped communication in accessing a remote file server [19].

In this paper, we propose, implement, and evaluate the TCP Server architecture in two different scenarios. The first scenario consists of using one or more dedicated processors to perform TCP 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, incurring 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. Finally, we study the tradeoffs between polling and interrupts for event notification in this environment.

In our second scenario, we offload the network processing to dedicated node(s) in a cluster-based server. The TCP server connects to the nodes running the server application through memory-mapped communication over a high-speed interconnect. We present and evaluate the design space of TCP offloading over memory-mapped communication using a user-level TCP server implementation over VIA. We present the performance gain for server applications using the MemNet API thus enabling a separation of the gains achieved from TCP/IP offloading, zero-copy implementation and asynchronous processing. Offloading the network processing to dedicated components has benefits that go beyond just performance, and increases modularity and ease of management of the server system.

This paper represents the first study to evaluate the benefits of TCP processing offloading in a comprehensive manner. Previous studies and products do not consider several important issues. First, they have not quantified the impact of offloading on the performance of network servers. We do so for different architectural scenarios. Second, previous work does not consider the role of the network programming interface in server performance. We demonstrate the gains achieved by server applications using a programming interface which allows the applications to exploit the benefits of the low latency communication between the host and the TCP Server.

Based on our experience and results, we conclude that offloading the network processing from the host processor using a *TCP Server* architecture can be beneficial for server performance in most scenarios.

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 and 5 describe the details of each architecture including an evaluation. Finally, section 7 presents our conclusions.

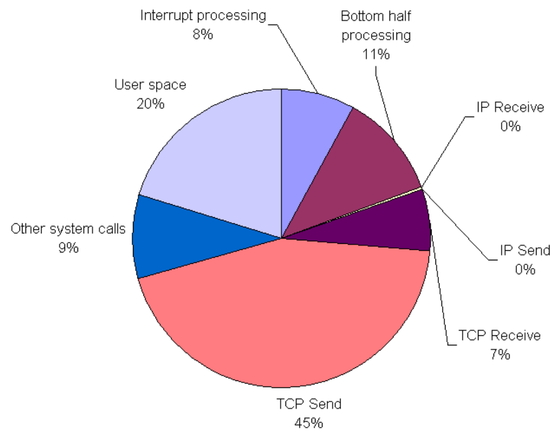

Figure 1. Apache Execution Time Breakdown

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 present experimental results in support of these statements, suggesting a need to offload networking functionality from a host.

To exemplify, 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. The TCP/IP processing for the `send` call takes 45% (including time spent making two copies of data, one from user space to kernel, another for cloning packets inside the kernel for potential retransmission). Interrupt processing (8%) includes the time to service NIC interrupts and setup DMA transfers. TCP receive (7%) is the time taken by the kernel to receive the packet, not including the time spent by the `recv` system call to copy the data into user space. Altogether, network processing, which includes TCP send/receive, interrupt processing, bottom half [1] processing, and IP send/receive take 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 believe that offloading TCP/IP processing from the host processor to a dedicated processor would help in improving server performance in two ways. First, by freeing up precious host processor cycles for the application. Second, by eliminating the harmful effects of *OS intrusion* [25] on the application execution.

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 aims to improve server performance by isolating the application from OS intrusion, and by removing the harmful effect of co-habitation of various OS services. The performance of applications using the TCP Server architecture heavily relies on the efficiency of the communication between the host and the TCP server, and on the efficiency of the socket programming interface used by the server application. A TCP server can execute the entire TCP processing or it can split the work with the application hosts.

Figure 2 presents two architectures for network servers: a tradtional architecture and an architecture based on TCP Servers. 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. As the goal of TCP/IP offloading is to save network processing overhead at the host, using a faster and lighter communication channel for tunneling is essential. In the TCP server implementations described in this paper, we use shared memory and memory-mapped communication for tunneling which are both non-intrusive communication solutions (with zero overhead on the remote side).

The TCP Server architecture enables several key

[1] In Linux, "bottom half" denotes the "soft interrupt" part of the interrupt processing. We distinguish it from the strictly asynchronous servicing of a hardware interrupt, which schedules it for subsequent processing.

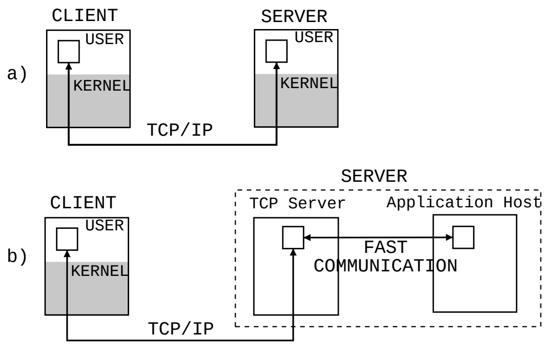

Figure 2. TCP Server Architecture

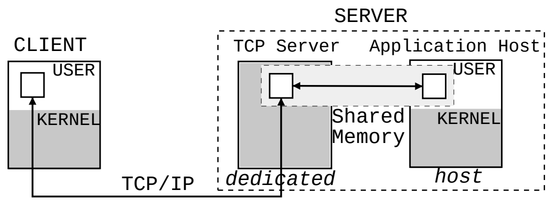

Figure 3. TCP Server in an SMP-based server

features which can significantly impact the overall performance of the server. In what follows, we will briefly discuss these features, their impact on the application programming interface and performance.

1. **Kernel Bypassing.** To achieve good performance, the communication with the TCP server must be done from user-space (application) directly, without involving the host OS kernel in the common case. This can be done by establishing a *socket channel* between the application and the TCP server for each open socket. In the SMP server case, the socket channel is implemented using shared memory queues, whereas in a cluster-based server the channel reduces to a memory-mapped communication channel (VIA/IB channel).

2. **Asynchronous Socket Calls**. By using asynchronous socket calls, the application can exploit maximum gain from using the TCP Server architecture. First, this allows for maximum overlapping between the TCP processing of the socket call and the application execution. Second, using asynchronous calls gives the server system scope to avoid context switches in the critical path whenever possible.

3. **Avoiding Interrupts**. Since the TCP server exclusively executes TCP processing, interrupts can be easily and beneficially replaced with polling on the TCP server. We evaluated both methods for the SMP-based server and found that replacing interrupts with polling is indeed beneficial. 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.

4. **Processing Ahead**. Decoupling the TCP/IP processing from the application enables a few optimizations to server processing. The TCP server can execute certain operations ahead of time, before they are actually requested by the application. The operations that can be eagerly performed and can provide performance benefits are the `accept` and `receive` system calls.

5. **Direct Communication with File Server**. In a multi-tier architecture that uses remote file systems for data storage, a TCP server can be instructed to perform direct communication with the file server. This means that certain files which the application does not want to cache in the host memory can be transferred directly from the file server to the TCP server. This transfer can be done securely if the host OS passes the socket channel (as a capability) to the file server which in turn uses it to write the file data onto the socket. This is particularly appealing for cluster-based servers over VIA if both DAFS servers and TCP servers are used, since both understand VI channels.

4 TCP Server Implementations

In this section, we present the implementation of the TCP Server architecture in two different scenarios:

- In a *symmetric multiprocessor(SMP) server*, the TCP server is implemented by dedicating a subset of the processors for in-kernel TCP processing.
- In a *cluster-based server*, the TCP server is implemented by dedicating a subset of nodes to TCP processing. A fast, low-overhead memory-mapped communication architecture such as VIA or Infiniband is used for intra-server communication.

4.1 TCP Server in SMP-based Servers

We partition the set of processors in an SMP-based server into *host* and *dedicated* processors as shown in

Figure 3. 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.

System Operation: 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 carries out the bottom half of the send processing, which also includes the IP send processing and setting up a DMA to the NIC.

The receive events, which are asynchronous, are routed to the TCP server, which does the entire bottom half processing from servicing a hardware interrupt, to IP receive and the bottom half TCP receive, where the protocol stack queues the data in the socket buffers. On a receive call from the application on the host processor, these buffers are copied to the user space.

Offloading TCP/IP functionality: We identify four distinct components of TCP/IP functionality that can be offloaded to dedicated processors: *(i)* the interrupt and bottom half (software interrupt) processing, *(ii)* the asynchronous receive processing (after the bottom half), *(iii)* the TCP/IP send functionality executed in the context of the process (after a system call), and *(iv)* the portion of the TCP/IP send processing *after* copying the data to kernel buffers.

In most scenarios, *(i), (ii)* cannot be offloaded independent of each other without incurring excessive overhead. In the offloading choices available for *(iii)* and *(iv)*, we can offload *(iv)* without changing the socket API semantics. While *(iv)* alone has some performance benefit, it does not alleviate the copy to the kernel buffers being done at the application processor. The offloading of *(iii)* subsumes that of *(iv)*, but requires a co-operating application to call an asynchronous `send` which returns immediately but the application cannot re-use the buffer until the `send` is done, i.e. TCP server has copied data to the kernel buffers. We discuss each of the design choices and the issues involved in implementing them, in the following sections.

Offloading interrupts and receive processing: Receive processing is asynchronous and executes in the context of an external interrupt. The network processors assume responsibility of receive processing after receiving the interrupt. This includes the interrupt

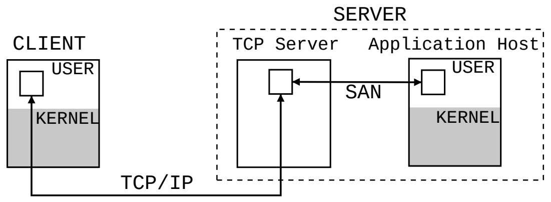

Figure 4. TCP Server in cluster-based servers

handling, bottom half functionality, IP, and TCP receive processing where the system copies the received data into the receive buffers of the socket. The effect of dedicating processors for receive processing is not limited to isolating the host from interrupts generated for TCP/IP processing.

Polling on the network interface has been suggested as an alternative mechanism to alleviate this problem [20, 22]. The inefficiency of polling has often been cited as a reason not to use it exclusively(instead of the interrupt mechanism) [5, 20, 22]. Our model, 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.

Offloading TCP send processing: TCP send requests are issued by the application. The OS copies the send buffer to kernel buffers, to prevent it from being overwritten before being sent out. A second copy is needed to allow TCP to retransmit the data in case of an error.

We propose two mechanisms to offload the TCP send processing to the dedicated processors. First, we provide a mechanism to offload the TCP send processing without any support from the application. In this case, we offload the send processing to the dedicated processor after the copy to the kernel buffers has been made in the context of the application. Second, we provide a send mechanism to avoid copying in the application processor TCP/IP stack including the copy to the dedicated processor, with co-operation from the application. In this case, the server asynchronously notifies the application about the completion of the send, after the data has been copied to or acknowledged at the TCP server. The system assumes that applications using asynchronous send, check for completion of the send before reusing the buffer.

4.2 TCP Server in Cluster-based Servers

In a cluster-based server, the application host and the TCP server are connected by a VIA-based SAN as

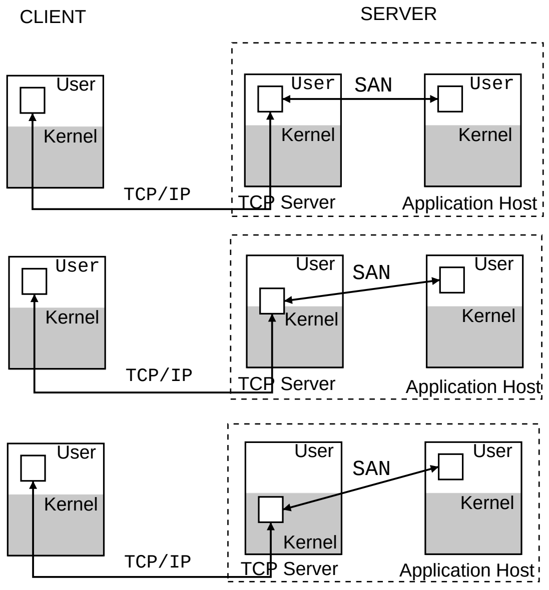

Figure 5. TCP Server Design Alternatives

shown in Figure 4. 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 VI channels. The socket call interface is implemented as a user-level communication library on the application host. This library manages and maintains VIs on the host. With this library, a socket call is tunneled across the SAN to the TCP server. On the TCP Server, a *socket provider* module interprets the call and performs the corresponding socket operation.

Design Alternatives: Figure 5 shows the various design alternatives to implement the TCP server. In the first two alternatives, the *socket provider* module is implemented in user space on the TCP server. In the first alternative, *socket provider* uses the standard Linux socket implementation to implement socket operations. With this approach, we still pay the complete cost of socket operations but on the TCP Server instead of the host. The second alternative avoids the copy between user-space and kernel-space by sharing the data buffers between the user-space *socket provider* and the kernel. The third alternative avoids the copy between user-space and the kernel by using an in-kernel *socket provider*. We used the first alternative for the prototype described in this paper. We are currently working on an optimized implementation based on the second and third alternatives.

Programming Interface: Server applications use the MemNet API [31] to access the networking subsystem in our prototype.

The MemNet API allows applications to perform sends and receives both synchronously and asynchronously. The send/receive primitives provided by the MemNet API (`sync_send, sync_recv, async_send and async_recv`) 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.

Prototype 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 with each VI channel, a registered memory region 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.

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. To guarantee correct operation, buffers used in send should not be overwritten until the entire buffer is sent to 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.

Optimizations: Our prototype also includes two optimizations to improve the performance of server applications.

- **Eager Receive** is an optimization to 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 eagerly 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 *socket provider* eagerly accepts connections upto a predetermined maximum. When the application issues an `accept`, one of the previously accepted connections is returned. We expect this optimization to reduce the processing time for the `accept` primitive.

5 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.

5.1 Evaluation of SMP-based Server Architectures

In this section we present the evaluation of the TCP offloading approach in an SMP-based TCP server architecture. In our prototype, we modified the Linux-2.4.16 networking stack to separate the *lower half* of both send(TCP send after copying in the kernel, IP send and device access routines), and receive processing(Interrupt service routines, software interrupt handlers, IP receive and copying to the socket buffer) from the *upper half*(system call interface, copying to and from the kernel buffers) of the kernel. We execute the lower half on the TCP server, which is a subset of processors in an SMP system. The upper half executes on the host processors. The TCP server processors are dedicated to network processing and do not execute any user level code. We isolate the asynchronous events(interrupts) from the host processors by routing them to the TCP server nodes using the IO/APIC interrupt routing mechanism.

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 use interrupts and polling as the alternative event notification mechanisms. For both polling and interrupts, we vary the amount of processing offloaded to the dedicated processors, *(i)* The dedicated processors do not participate in any send processing, and *(ii)* The lower half of the send processing is also offloaded to the dedicated processors. In this case, the host processor is responsible for user-level processing and for transfer of data between user space and kernel space. While in the first case, the TCP server is primarily responsible for asynchronous event handling, in the second case, the TCP server also participates in the network send processing . This gives us four different configurations as combinations of the alternatives.

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.

We evaluate these systems on two hardware configurations *(i)* a 300 MHz Intel Pentium-based 2-way SMP system with 512 MB DRAM and 256 KB L2

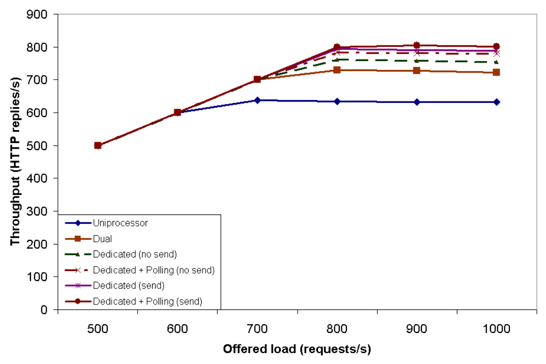

Figure 6. Throughput for a simple web server on a 2-Way SMP system using HTTP/1.0.

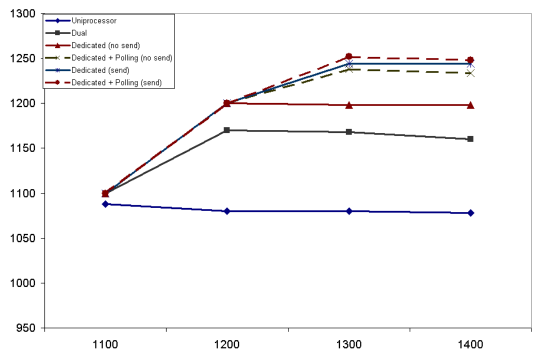

Figure 7. Throughput for a simple web server on a 2-Way SMP system using HTTP 1.1

cache and *(ii)* a 550 MHz Intel Xeon-based 4-way SMP system with 1GB DRAM, 1MB L2 cache. Both configurations used a 3Com 996-BT gigabit Ethernet adapter.

Our experiments reveal that dedicating processors to asynchronous event handling improves the performace of a typical web server upto 30%. 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. We describe the results for the 2-way case in Section 5.1.1 and the 4-way case in Section 5.2.

5.1.1 Experimental results for 2-way SMP

We studied the performance of a simple multithreaded web server on a 2-way SMP system running different configurations of TCP servers described above. We used a synthetic trace(Synthetic) in which 16-KByte files are requested in a way that the files are unlikely to be found in the L2 hardware cache. The characteristics of the trace are shown in Table 1. We used httperf [24] as the client benchmarking tool to generate the required workloads. 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 1.1 protocols to measure server performance with this synthetic workload.

With the 2-way SMP machines, since we are limited by the number of processors, we can evaluate only two of the three dimensions of our design space. We evaluate the impact of different dedicated processor set sizes in Section 5.2 using 4-way SMP machines.

Figure 6 shows the throughput for the simple web server for different kernel configurations using the HTTP/ 1.0 protocol and Figure 7 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 upto 26% 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 significant improvement (about 15%). In both cases, we can see that the offloading of send processing helps but only to a limited extent, the major performance benefit is due to the removal of asynchronous network events from the host processor. This behaviour is due to the dedicated processor saturating before the host processor and becoming the bottleneck in the system. We show a more detailed analysis of this pattern in Section 5.2.

5.2 Experimental Results for 4-way SMP

We used Apache 1.3.20 web server as a sample server application for the 4-way SMP configurations. We used the default configuration for Apache which had a pre-forked set of five server processes and the maximum number of processes in the pool was limited to 150.

We used sclients [6] as the client program to generate the requests for the server. The clients request files according to a trace at the maximum rate a server can sustain. 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

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 1. Main characteristics of WWW server traces

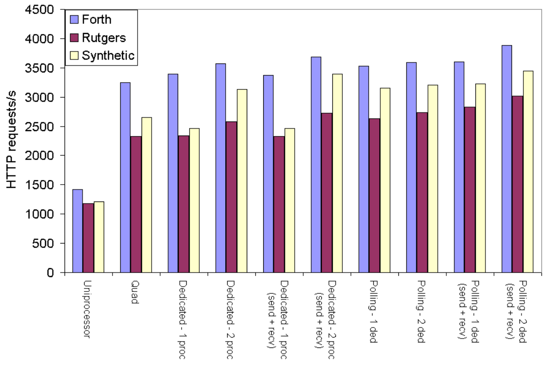

Figure 8. Throughput for Apache on a 4-Way SMP Server

days of March 2000. Synthetic is a synthetic trace in which 16-KByte files are requested in such a way that the files are unlikely to be found in the L2 hardware caches. Table 1 describes the main characteristics of these traces.

In Figure 8, we show the throughput obtained by the SMP server running on different kernel configurations. We plot the performance using the three traces for ten different configurations. Our first configuration is the uniprocessor system *Uniproc*. In the second configuration, *Quad*, the server runs on all four processors which also do network processing. We present results for this configuration as a basis for comparison. The next four configurations assume different splits of the network processing. *Dedicated 1 proc* and *Dedicated 2 proc* use one and two dedicated processors, respectively, for receive processing and network interrupts. In *Dedicated 1 proc (send+recv)* and *Dedicated 2 proc (send+recv)*, the processing of both send and receive operations is performed on the dedicated processor(s). The remaining configurations replace interrupts with polling as the mechanism for network event notification on the dedicated processors.

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 and hardware cache behaviors are different. Another interesting observation is that dedicating two processors to network processing is always better than dedicating only one. Offloading the send processing and polling, in particular, are only really beneficial when two processors are dedicated to the network processing. Overall, we can see that offloading the network processing can achieve improvemtents 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.

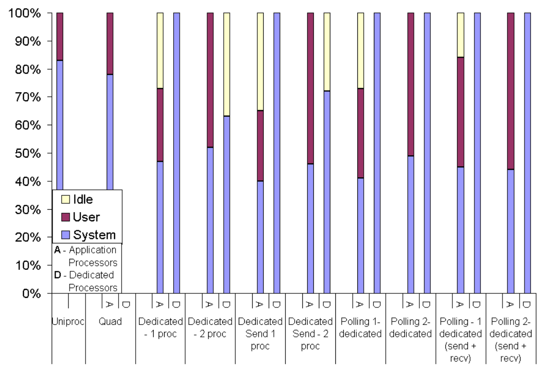

Figure 9. CPU usage for Apache on a 4-Way SMP Server

Figure 9 explains these high-level results. The figure depicts 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.

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 dedicated 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 polling configurations with two network processors do not show any idle time for the network processors. The reason is that we categorize their blocked 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 working on a system that performs such load balancing.

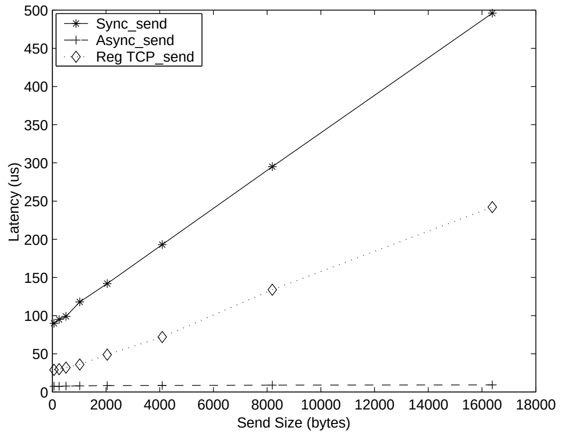

Figure 10. Cost of `send`

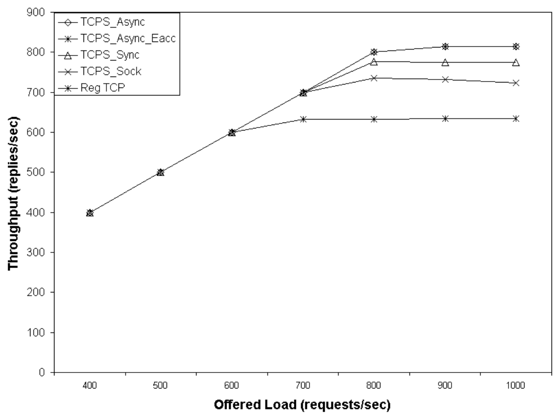

Figure 11. Web server throughput with HTTP/1.0 using cluster-based TCP Server

5.3 Cluster-based TCP Server Evaluation

In this section, we evaluate the TCP Server architecture for clusters. We developed a prototype using PCs connected by a VIA-based SAN. We used two 300 MHz Pentium II PCs, one each for the application host and the TCP server, 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. Both the host and the TCP server run Linux-2.4.16.

In our prototype, each socket call at the host is tunneled through VI channels to the TCP Server. In Figure 10, we compare the latency perceived by applications for synchronous and asynchronous sends in our prototype with the latency of send in a traditional system (Reg TCP_send), for a stream socket. We chose the send primitive as the send plays an important role in server applications like web servers.

The cost of `async_send` ($< 8\mu$s) is close to the cost of posting a send on the VI channel. However, the `sync_send` is expensive since it includes the cost of tunneling over the SAN and the cost of a traditional socket send at the TCP Server. `async_send` allows applications to hide the latency of the send by returning to the application immediately after the send is posted on the VI channel. The reduced latency of our asynchronous send implementations show scope for improvement in performance for server applications using these implementations.

5.3.1 Web Server Performance

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 (*TCPS_Sock*) and using the primitives provided by the MemNet API (*TCPS_Sync* and *TCPS_Async*) which require buffers used in communication to be pre-registered. In *TCP_Sync*, the web server implementation uses the `sync_send` primitive and in *TCP_ASync*, it uses the `async_send` primitive. We also present the performance of the web server using a standalone Linux host-based socket implementation (*Reg TCP*) for comparison.

Workload: The workload used for HTTP/1.0 consists of requests for 16-KByte static files, making sure that the requested file is not available in the L2 hardware cache. We used httperf [24] as the client benchmarking tool to generate the required workloads. We used a standalone PC with an unmodified Linux socket implementation for the client. We present the performance analysis for this synthetic workload using HTTP/1.0 and HTTP/1.1.

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 *Reg TCP* saturates at an offered load of 700 requests/second. The web server shows an improvement of 15% in performance with *TCPS_Sock* over *Reg TCP*. Using the synchronous primitives (*TCPS_Sync*), the web server is able to achieve a performance improvement of 22%. *TCPS_Async* shows a performance gain of about 30% with the web server using asynchronous primitives like `async_send`. *TCPS_Sync* shows a gain in performance due to offloading of network sends to the TCP Server. *TCPS_Async* in addition, allows a better pipelining of network sends and helps the application overlap the latency of offload-

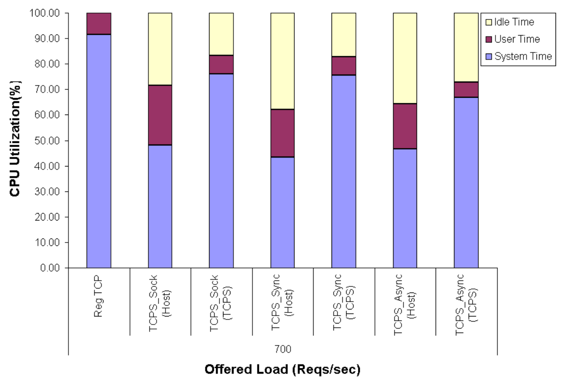

Figure 12. CPU usage for web server using HTTP/1.0

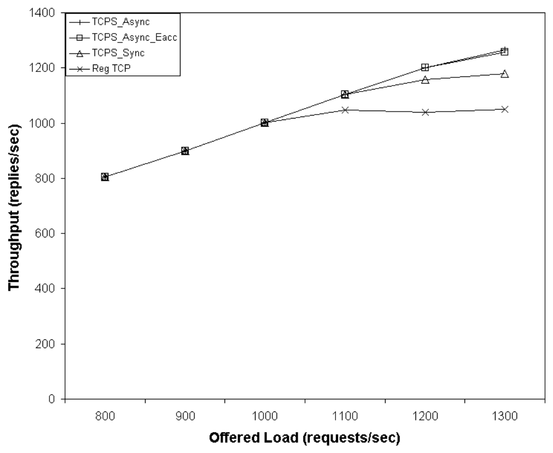

Figure 13. Web server throughput with HTTP/1.1

ing the send primitive over the SAN with computation at the host. *TCPS_Async_Eacc* includes the *Eager Accept* optimization in addition to *TCPS_Async*. This provided no additional gain since it is not the connection time, but the actual request processing time that dominates the network processing.

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.

In Figure 12, we present the CPU utilization on the application host (and TCP Server) for various systems, for the load at which *Reg TCP* saturates. At this load, the host CPU saturates for *Reg TCP* whereas the *TCPS_Sync (Host)* and *TCPS_Async (Host)* have about 40% idle time. With *TCPS_Sock*, since the web server uses only the traditional socket based API, it does not pre-register buffers used in communication. As a result, copies take up CPU time and reduce the idle time in *TCPS_Sock (Host)* to 29%. For *TCPS_Sock*, *TCPS_Sync* and *TCPS_Async*, we also present the CPU utilization on the TCP Server (*TCPS_Sock (TCPS) TCPS_Sync (TCPS) and TCPS_Async (TCPS)*) to show that the entire TCP/IP processing overhead has been shifted to the TCP Server in these systems. 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 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 *TCPS_Sync* is about 12%, and by *TCPS_Async* is 20%, over that of *Reg TCP*. 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*, do not improve throughput beyond that of *TCPS_Async*. 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 (either statically or dynamically), by using a faster TCP server, or by using multiple TCP servers per application node. We are currently investigating these approaches.

6 Related Work

OS mechanisms and policies specifically tailored for servers have been proposed in [7, 12, 29]. 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*, as described by Mogul and Ramakrishna [22]. Several researchers suggest the use of polling on the system to prevent receive livelock and for high performance [34, 20]. Aron and Druschel in [5] use the soft timer mechanism to poll on the network interface. The idea is extended in Piglet [26], where the application is isolated from the asynchronous event handling using a dedicated polling processor in a multiprocessor.

In the Communication Services Platform (CSP) [33] 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.

Intelligent network interfaces [27] have been studied, but mostly for cluster interconnects in distributed shared memory [15] or distributed file systems [4]. Recently released network interface cards have been equipped with hardware support to offload the TCP/IP protocol processing from the host [2, 3, 10, 14, 18, 36]. 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. Modeling and simulation were used in [9] to analyze a range of scenarios, from providing conventional servers with high I/O bandwidth, to modifying servers to exploit user-level I/O and direct device-to-device communication, and offloading file system and networking functions from the host to intelligent devices.

QPIP [8] 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 [8] was limited to communication between QP-aware applications at both endpoints over a SAN.

Sockets Direct Protocol (SDP) [30] originally developed by Microsoft 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 [35] 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 exploit features like RDMA, available in interconnect technologies like VIA [13] and Infiniband to provide a transport which includes remote memory semantics. 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. Unlike Piglet or CSP, we study the effect of such separation of functionality for the server systems under real server application workloads.

7 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 demand substantial computing resources for complete offloading. For a complete TCP/IP offloading to intelligent devices, the device has to be computationally powerful to outperform traditional architectures. (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 architecture, implementing dynamic load balancing between processors of different classes and implementing an optimized TCP Server for the cluster-based scenario.

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] 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.

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

[7] 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.

[8] 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.

[9] E. V. Carrera, M. Rangarajan, R. Bianchini, and L. Iftode. Impact of Next-Generation I/O Architectures on the Design and Performance of Network Servers. In *Proc. of the Workshop on Novel Uses of System Area Networks, SAN-1* , February 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] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

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

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

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

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

[20] 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.

[21] M. Thadani and Y. Khalidi. An efficient zero-copy I/O framework for UNIX, 1995.

[22] 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.

[23] 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.

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

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

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

[27] Myricom: Creators of myrinet. http://www.myri.com.

[28] 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.

[29] 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.

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

[31] 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-481, March 2002.

[32] RapidIO Trade Association. http://www.rapidio.org.

[33] 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 *Proceedings of 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

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

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

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

[37] 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.