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

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

March 29, 2002

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 three *TCP Server* architectures: (1) a dedicated network processor on a symmetric multiprocessor (SMP) server, (2) a dedicated node on a cluster-based server built around a memory-mapped communication interconnect such as VIA, and (3) an intelligent network interface in a cluster of intelligent devices with a switch-based I/O interconnect such as Infiniband.

Based on our experience and results, we can draw a few important conclusions: (i) an SMP-based approach to TCP servers is more efficient than a cluster-based one, given the lower overheads and the slightly more even division of labor between application and network processors of the former approach, (ii) the benefits of SMP and cluster-based TCP servers reach 18% and 30% respectively, in the scenarios we studied, (iii) the simulated results show greater gains of up to 45% for a cluster of intelligent devices, and (iv) it is clear that greater gains are possible only if we perform dynamic load balancing between hosts and the TCP servers.

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 [31] 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 [12, 26].

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 [41, 25]. 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 [5].

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, 11, 14, 19, 24, 40] and *(ii)* replacing the expensive TCP/IP processing with a lightweight, more efficient transport protocol using user-level and memory-to-memory communication based on standards such as VIA [13] and Infiniband (IB) [18, 35]. Both approaches are currently competing to become the supporting technology for remote storage access in multi-tier data centers [21, 33]. 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 [29].

Our work aims to understand the design, imple-

*Currently with the Department of Computer Science, University of Maryland, College Park, MD 20742

mentation, and performance of server architectures that rely on TCP/IP offloading for client-server communication. Our approach consists in 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 [18, 34]. 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. A similar technique is used by the Direct Access File System (DAFS) standard which exploits memory-mapped communication in accessing a remote file server [22].

In this paper, we propose, implement, and evaluate three different *TCP Server* architectures. The first architecture consists of using one or more dedicated processors to perform TCP processing in a Symmetric Multiprocessor (SMP) server. This architecture is similar to the one proposed in Piglet [29]. 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.

Our second *TCP Server* architecture uses a dedicated node in a cluster-based server to offload the network processing. The TCP server connects to the nodes running the server application through memory-mapped communication over a high-speed interconnect. This architecture is similar to the one proposed in CSP [35]. We present and evaluate the design space of TCP offloading over memory-mapped communication using a user-level TCP server implementation over VIA.

The third architecture corresponds to offloading the TCP processing to one or more intelligent NICs over an I/O switch with memory-to-memory communication support. We extend this architecture to define a cluster of intelligent devices (CID), as a system in which intelligent devices are connected to the host processor(s) by a switch-based I/O interconnect. We used simulations to evaluate the impact of offloading on the performance of a web server for various host processor speeds and present the server performance as a function of the processing power available on the intelligent NIC. The simulations allow us to quickly explore the design-performance space for the proposed TCP server architectures by extrapolating the experimental results.

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 several different architectural scenarios. Second, previous work often assumed that an intelligent NIC would only perform simple protocol tasks, such as packet aggregation. The advent of high-performance, low-power microprocessors opens the possibility of more powerful intelligent NICs [20]. We demonstrate how the extra power can be exploited effectively and how much performance improvement results.

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. The next section describes our motivation for this work in detail. Section 3 provides an overview of the *TCP Server* architecture. Sections 4.1, 4.2, and 4.3 describe the details of each architecture and their most important results. Finally, section 7 draws 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 the 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.

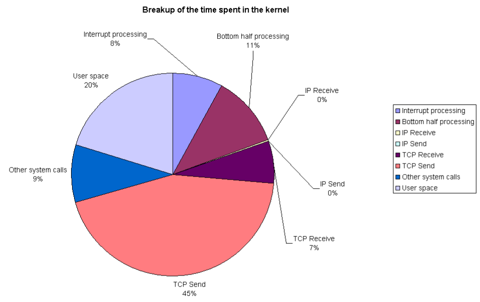

Figure 1: Apache Execution Time Breakdown

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.

This data led us to believe that offloading TCP processing from the host processor to a TCP server would help in improving server performance. First, by freeing up precious host processor cycles for the application. Second, by eliminating the harmful effects of *OS intrusion* [28] on the application execution.

3 TCP Server Architecture

A 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 the TCP server 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.

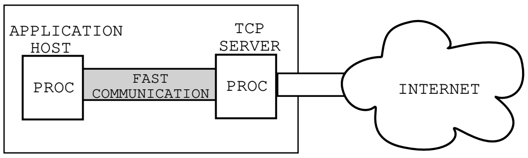

Figure 2: TCP Server Architecture

The basic TCP server architecture is presented in Figure 2. 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 in remote procedure calls similar to the way a remote file system such as NFS operates. As the goal of TCP offloading is to save network processing overhead, 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 design of the TCP server addresses several key issues which can significantly impact the overall performance of the server. In what follows we will briefly discuss these design issues, 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. To ensure protection between applications, the socket channel is created by the host OS kernel during the socket call (which is still implemented as a system call). In the multiprocessor server case, the socket channel is implemented as a shared memory region, whereas in a cluster-based server the channel reduces to a memory-mapped communication channel (VIA/IB channel). In the cluster-based server case, the application must also pre-register its send buffers with the memory-mapped communication adapter.

2. **Asynchronous Socket Calls**. By using asynchronous socket calls, the application can significantly improve the efficiency of using the TCP server. First, this allows for maximum overlapping between the TCP processing of the socket

[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.

call and the application execution. Second, by using asynchronous calls, the server can avoid context switches whenever this is 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 and found that replacing interrupts with polling is indeed beneficial. However, the frequency of polling must be carefully controlled, as a very high a 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. **Avoiding Data Copying**. With asynchronous system calls, the TCP server can avoid the double copying performed in the traditional TCP kernel implementation of the send operation. To achieve this, the application must tolerate the wait for 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 copy on a send operation. For retransmission, the TCP server can read the data again from the application send buffer using non-intrusive communication (shared memory or remote read operations on memory-mapped communication). Send buffer registration guarantees pinning to physical memory which is absolutely necessary for the TCP server to retrieve the send buffer at retransmission time.

5. **Processing Ahead**. 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 significant performance benefits are the `accept` and `receive` system calls.

6. **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 solution is particularly appealing for cluster-based servers over VIA if both DAFS servers and TCP servers are used, since both understand VI channels. We plan to evaluate this in the final version of the paper if it's accepted.

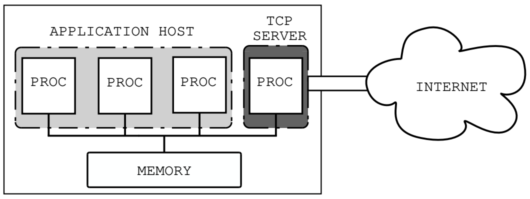

Figure 3: TCP Server in a SMP-based architecture

4 TCP Server Implementations

In this section, we present the design and evaluation of three TCP Server architectures.

- 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. In our current prototype, we implemented the TCP server as a user-process on the networking-dedicated nodes.

- In the next-generation of servers built as a *cluster of intelligent devices over switched-based I/O*, such as Infiniband, the TCP server can be implemented in the intelligent network interface (I-NIC). We study this architecture through simulations.

4.1 TCP Server for SMPs

We partition the set of processors in an SMP-based server into *host* and *network dedicated* processors as shown in Figure 3. The network processors execute the TCP server exclusively. The TCP server executes a tight loop in the kernel context on each dedicated network processor. Network generated interrupts are routed to the dedicated processors. The communication between the application and the TCP server is through queues in shared memory, where the application places the offloading requests and the TCP server retrieves and services them.

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 in 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 them in the following sections.

4.1.1 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 handling, bottom half functionality, IP, and TCP receive processing where the system copies the received data in 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 [25, 23]. The inefficiency of polling has often been cited as a reason not to use it exclusively(instead of the interrupt mechanism) [23, 25, 6]. Our model, where dedicated processors execute in an infinite loop, 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 Offloading TCP send processing

TCP send requests are issued by the application. The OS copies the send buffer to 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 mechanism to avoid copying in the application processor TCP/IP stack including the copy to the dedicated processor, with co-operation from the application. This is possible as the send completes only when the server notifies the application. This notification is sent asynchronously after the data has been copied to or acknowledged at the TCP server. If the application uses asynchronous send, it has to check for completion of the send before reusing the buffer.

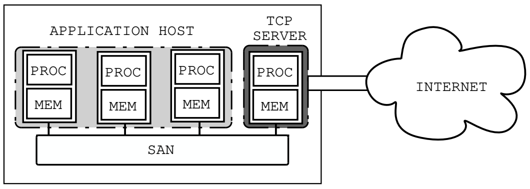

Figure 4: TCP Server in cluster-based architectures

4.2 TCP Server in Cluster-based Architecture

The cluster-based TCP Server architecture consists of a cluster of PCs connected by a VIA-based SAN Figure 4.

In our implementation, we have a user-level TCP server running on a dedicated node in the cluster. The VIA-based communication layer provides the socket channels between the host PC and the TCP server. The TCP server node acts as the network endpoint for the outside world and the network data is transferred between the host and the TCP server node across SAN using VIA channels. This view is similar to a multi-tier architecture for cluster-based servers and has been discussed before in CSP [35]

The user-level TCP server process running on the network node manages the communication between the host and the TCP server node. As creating VIs and registering the memory required for the VI communication are expensive, the host creates a pool of VIs at startup and requests connections on them from the TCP server. The main thread of the TCP server responds to VI connection requests from the host and accepts or rejects VI connections 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 call interface is implemented as a user-level communication library. This library manages and maintains VIs at the host side. With this library, a socket call is tunneled across SAN to the

TCP server. The TCP server interprets the calls and performs the networking operation using the Berkeley socket library. The mapping of a socket to a VI and its associated memory regions is maintained for the lifetime of the socket. Since the host memory regions will be associated with the socket during its life time, the TCP server can do RDMA writes as required and this facilitates intelligent processing by the TCP server.

In what follows, we describe several implementations of the TCP server for cluster-based servers that we evaluated for this study.

- **Split-TCP** implements a synchronous offloading of network processing to the TCP server. When the application makes a socket library call on the host, our library tunnels the socket call parameters to the TCP server and blocks waiting for a response. The TCP server processes the socket call and responds to the host, and the call then returns to the application. This approach does not take advantage of any parallelism inherent in the communication or the capabilities of the VIA-based interconnect to improve performance. We use this model as a baseline and optimize it to make our TCP server system outperform the traditional single machine networking stack.

- **AsyncSend** implements an optimization to the network send processing. The send is asynchronous and returns to the application as soon as the arguments to the send routine are tunneled to the TCP server node. The application proceeds as soon as the data to be sent is transmitted over the VIA channel. This is analogous to what happens in traditional UNIX systems where the send system call returns to the application immediately after the data to be sent is copied into the kernel buffers. The TCP server on receiving the send data, issues the `send` call and returns the result to the host using RDMA. Each asynchronous send call at the host first checks for errors from previous sends before proceeding. An RDMA based flow control mechanism controls the number of requests that a host can pipeline to the TCP server.

- **Eager Receive** implements an optimization to the network receive processing. The TCP server eagerly performs *recv* operations on behalf of the host and when the host application issues a `recv` call, data is transferred from the TCP server node to the host. The TCP server posts receive eagerly for a number of bytes, and, depending on the consumption speed of data by the host, continues with further eager receive processing. The TCP server uses the *poll* system call to verify if any data has arrived for that socket connection before issuing an eager recv. The TCP server could push the received data to the host, but this trades off the latency of a roundtrip over the SAN for an additional copy at the host from the received data buffers to the user buffers. Only a pull based Eager Receive is currently implemented and analyzed.

- **Eager Accept** implements an optimization to the connection processing. While the `send` and `recv` optimizations are done on a per socket basis, the Eager Accept is a system wide optimization. A dedicated thread of the TCP server eagerly accepts connections upto a pre-determined maximum. When the host issues an `accept` one of the previously accepted connections is returned to the host. This optimization will reduce the processing time for the `accept` call in the critical path. The TCP server can also use filters to selectively accept only certain connections on behalf of the host.

- **Setup With Accept** Associating a socket with a VI for the entire lifetime of the connection may affect concurrency and hence in the normal case, the communication library does not associate the VI to the socket at the time of accepting a new connection. Instead, it postpones this association to the time of the first socket call(which is usually a `recv` for server applications). With this model, an additional round trip is required for the **AsyncSend** and **EagerRecv** implementations, to inform the TCP server about the host RDMA regions. The *Setup With Accept* does the association of socket to VI at the time of the `accept` call and exports the host RDMA regions along with the `accept`.

4.3 TCP Server in a Cluster of Intelligent Devices

In the previous section, we described an architecture where the TCP/IP processing was offloaded from the host processor to a dedicated processor over a SAN. In this section, we extend the study to evaluate server performance with TCP/IP offloading from the host processor to an intelligent NIC (I-NIC), assuming that the host and the I-NIC are interconnected by an Infiniband switch, as shown in Figure 5.

In this scenario, the devices are considered to be “intelligent”, i.e., each device has a programmable processor and local memory. Thus, each NIC becomes a TCP server in this new architecture.

Our idea is to use the I-NIC to maintain the soft-state of the TCP/IP connections and perform most of

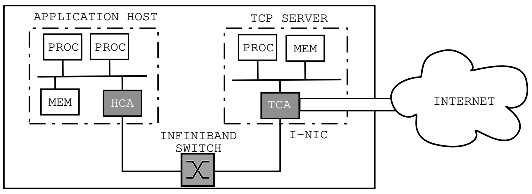

Figure 5: TCP Server in a Cluster of Intelligent Devices

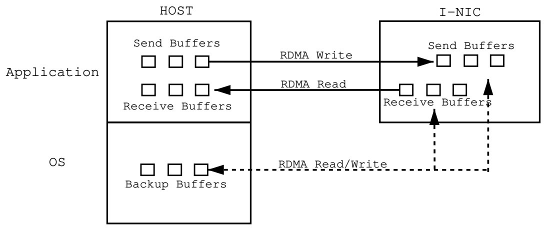

Figure 6: Interaction between HOST and I-NIC

the TCP/IP processing on the NIC. Figure 6 presents the mechanism for direct interaction between the application and the I-NIC. We use the local memory of the I-NIC to cache send and receive buffers of open connections. When a connection is established, send and receive buffers are allocated for that connection, mapped on the I-NIC, and handles to these memory regions are returned to the host. These handles are used on subsequent send/receive calls for remote DMA transfers of data associated with that connection. The backup buffers are used to save copies of send buffers at the host for possible retransmission, when the I-NIC runs out of buffer space. The backup buffers are also used to cache the data received at the NIC until the application performs the receive operation.

Each open connection is associated with a memory-mapped channel between the host and the I-NIC. During a message send, the message is transferred directly from user-space to a send buffer at the interface. A message receive is first buffered at the network interface and then copied directly to user-space at the host.

In Section 5.3, we evaluate the performance impact of TCP/IP offloading for this architecture by using simulations.

5 TCP Server Evaluation

In this section, we present the performance impact of the TCP Server architecture in three different scenarios.

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 the WWW server traces

5.1 Evaluation of SMP-based Server Architectures

In this section we present the evaluation of the TCP offloading approach in an SMP-based architecture.

5.1.1 Experimental Setup

We modified the Linux-2.4.16 kernel to implement our TCP Server architecture and tested it on two configurations, *(i)* a 300 MHz Intel Pentium-based 2-way SMP system with 512 MB DRAM and 256 KB L2 cache and *(ii)* a 550 MHz Intel Xeon-based 4-way SMP system with 1GB DRAM, 1MB L2 cache. Both configurations had a 3Com 996-B gigabit Ethernet adapter. We used the Apache 1.3.20 Web server as a sample application for our system. 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 [7] as the client program for the 4-way case, and httperf [27] to generate the requests for the server. The clients request files according to a trace at a rate specified in the configuration. 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. The trace is organized in such a way that the files are unlikely to be found in the L2 hardware caches. We used Forth, Rutgers and Synthetic for the 4-way SMP configuration and Synthetic for the 2-way SMP configuration. Table 1 describes the main characteristics of these traces.

5.1.2 Experimental results for 2-way SMP

We studied the throughput with the 2-way SMP configuration for the Synthetic trace with varying offered load. Figure 7 shows the throughput for the Apache webserver for different kernel configurations. Our first configuration is the uniprocessor system *Uniproc*. In the second configuration, *Dual*, the server runs on both 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*

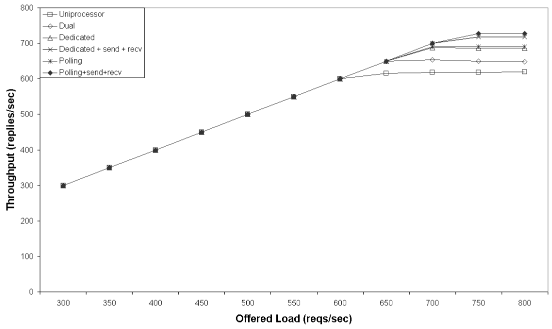

Figure 7: Throughput for Apache on a 2-Way SMP system, with a static workload.

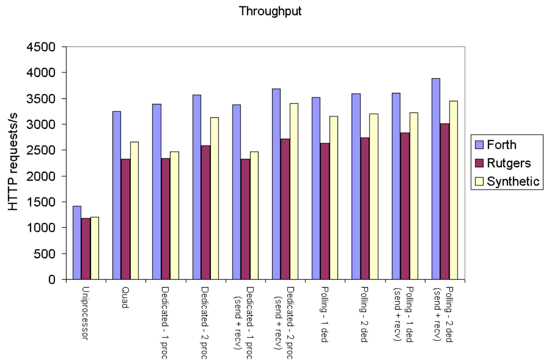

Figure 8: Throughput for Apache on a 4-Way SMP Server.

uses one dedicated processor, for receive processing and network interrupts. In *Dedicated+send+recv*, the processing of both send and receive operations is performed on the dedicated processor. The remaining configurations replace interrupts with polling as the mechanism for network event notification on the dedicated processors.

We observe that dedicating a processor for interrupt and receive processing helps improve throughput by 12% and performing the send processing as well on the dedicated processor increases the gain to 16%.

5.1.3 Experimental Results for 4-way SMP

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 widely 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 network processors and polling. This result demonstrates that this TCP server architecture can indeed provide significant performance gains.

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 further loading it 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 that 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.

5.2 Evaluation of Cluster-based Server Architectures

In this section, we discuss the results obtained with our implementation of TCP/IP offloading for clus-

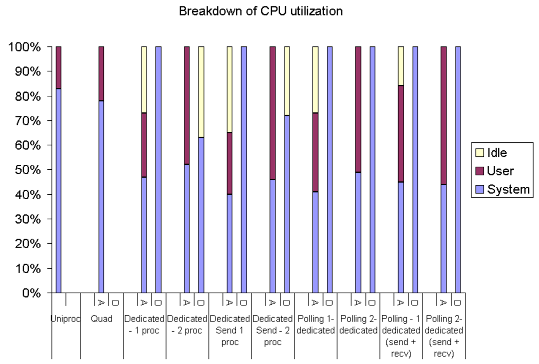

Figure 9: CPU usage for Apache on a 4-Way SMP Server

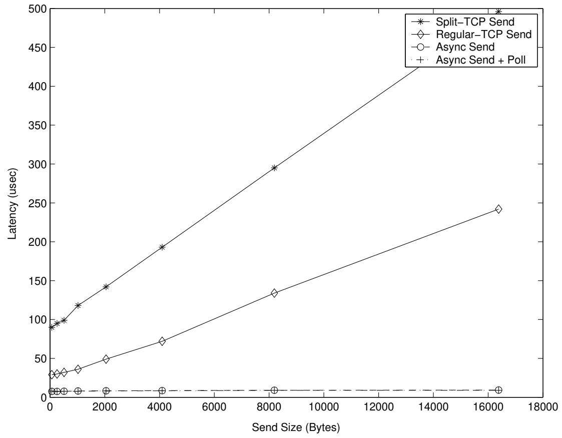

Figure 10: Cost of `send` for a cluster-based server

ters.

5.2.1 Experimental Setup

For the experiments with the cluster-based TCP server, we use two 300 MHz Pentium II PCs, a host and a TCP server, that communicate over 32-bit 33 MHz Giganet cLAN interfaces and an 8-port Giganet switch. Both the host and the TCP server nodes run Linux-2.4.16.

5.2.2 Network Operation Costs

In our TCP Server architecture, each socket call at the host incurs the additional cost of tunneling through VIA channels. Hence the application actually perceives an increased latency for each call. To give a sense of how this latency will affect the performance, we have analyzed the `send` library call, since this call is important in the context of network servers. Figure 10, shows the latency of the `send` call for various packet sizes.

In the figure, *Regular-TCP Send* refers to the native, non-offloaded `send` socket call and is used here as the basis for comparison. *Split-TCP Send* refers to our library implementation of the `send`, which incurs the additional overhead of the `VipRecvWait`, the blocking receive implementation provided by VIA.

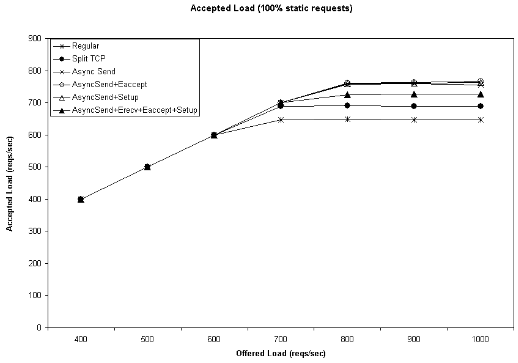

Figure 11: Throughput for a simple web server on a cluster-based architecture, with a static workload

Async Send refers to our **AsyncSend** optimization. The reduction in latency provides the motivation for **AsyncSend**. *Async Send + poll* tries to avoid the overhead of the `VipRecvWait` by polling instead of waiting on the VIA channels. The reduced latency of our asynchronous send implementations show scope for improvement in performance for servers using these implementations.

5.2.3 Performance Analysis

We analyze the performance of the TCP offloading across clusters using a simple multi-threaded http server. We use httperf [27] as the client benchmarking tool to generate the required workloads. We study two synthetic workloads using HTTP 1.0. The first workload consists of requests for 16-KByte static files, making sure that the requested file is not available in the L2 hardware cache. The second workload combines requests for static and dynamic content (at a ratio of 20% dynamic content to 80% static content) The dynamic content behavior is that suggested by the WebStone 2.0.1 [39] benchmark, i.e. the data returned is computed by making a call to a random number function for each byte returned. No threads or processes are created on the critical path of the dynamic content requests. Our results for both workloads compare the behavior of *Split-TCP* and the various optimizations discussed in Section 4.2 against *Regular-TCP*.

Static workload: Figure 11 presents the throughput of the server for the first workload, as a function of the offered load in requests/second. We show that *Split-TCP* shows an improvement of 7% in performance compared to *Regular-TCP*, with a higher rate of accepted requests at high offered loads. *AsyncSend* shows a 17% increase in throughput compared to *Regular-TCP*. The additional improvement achieved by *AsyncSend* is due to the fact that sends return faster to the application, and the host can pipeline network requests.

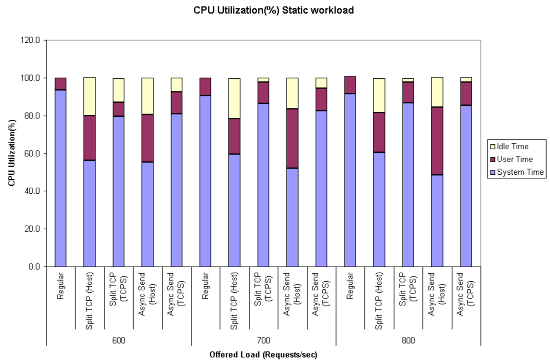

Figure 12: CPU usage for a simple web server on a cluster-based architecture, with static workloads

A combination of *AsyncSend + EAccept* exhibits the same behavior as that of *AsyncSend* since *EAccept* does not contribute in improving the `send` performance. The same can be said of *AsyncSend + Setup*, which removes one round trip across the VIA channels, from the critical path of the request processing.

ERecv alone does not help for this workload since the first `recv` on the socket is what triggers the eager receive processing, but in HTTP 1.0 there is only one request per connection. *ERecv + Setup* does not perform well and, in fact, suffers at high loads. The reason for this is that the socket to VI mapping is maintained for longer periods of time, so at high loads we do not have enough VIs to satisfy the requests.

Finally, contrary to our expectations, putting together all the optimizations *AsyncSend+ERecv+EAccept+Setup* does not provide the maximum gain. This is because of *ERecv*. After processing each request from the host, the TCP server attempts to do a `poll` system call to verify if there is data on the socket to be received.

Figure 12 provides better insight into the performance of the different systems by presenting the CPU utilization for the two nodes. It is interesting to note that, even before saturation point (load of about 600 reqs/sec), with *Regular-TCP*, the CPU does not have any idle time. In the case of *Split-TCP* and *AsyncSend*, the host has idle time available since it is the network processing at the TCP server that proves to be the bottleneck. The CPU usage on the TCP server for *Split-TCP* is very similar to that of *Regular-TCP*. The host processor, though not fully saturated, spends a considerable amount of time on system processing. The user time spent by the *AsyncSend* host is higher than that of *Split-TCP* since it is able to process more requests.

These results show that we can achieve improvements of upto 17% with this architecture for static workloads. Greater gains are not possible because the TCP server node is excessively loaded. In fact, this explains why some of our combined optimizations, e.g., *AsyncSend + EAccept*, do not improve throughput beyond that of *AsyncSend*. 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 moving some of the load back to the application node (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.

Combined workload: We also explored the same performance tradeoffs when the workload consists of a more realistic combination of static and dynamic content requests. Figure 13 presents the server throughput as a function of the offered load. The *Split-TCP* and *AsyncSend* systems saturate later than *Regular-TCP*. The gain in throughput for *Split-TCP* compared to *Regular-TCP* is about 11%, which is higher than the corresponding gain with the static workload. However, the gain provided by *AsyncSend* is about 18% and is only marginally higher than the gain found with the static workload. The reason for this is clear from Figure 14, which shows that at an offered load of about 500 reqs/sec, the host CPU is effectively saturated both for *Split-TCP* and *AsyncSend*. Hence, the network performance does not improve, even though a lot of idle time is still available at the TCP server node. We do not present results for the other optimization combinations because of the saturation of the host node.

These results show that we can achieve improvements in throughput of upto 18% with this architecture for the combined workload presented. Greater gains are not possible because the host node (instead of the TCP server node as before) is excessively loaded for this architecture/workload combination. We can alleviate this problem in similar ways as for the other workload, but focusing on reducing the load on the host node.

Overall, the message we can get from the results for the two workloads is that balanced configurations depend heavily on the particular characteristics of the workload. When these characteristics are fairly static and known in advance, a balanced system can be implemented. When either of these conditions does not hold, a dynamic scheme for balancing the load between host and TCP server nodes is required for ideal performance. We are currently working on the implementation of such a scheme.

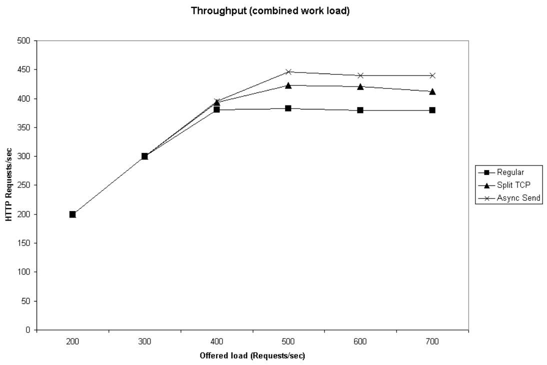

Figure 13: Throughput for a simple web server on a cluster-based architecture, with 20% dynamic work load

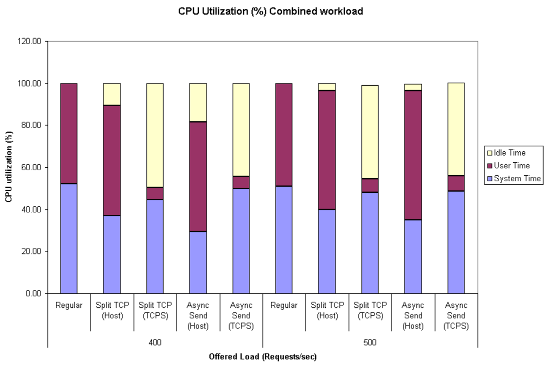

Figure 14: CPU usage for a simple web server on a cluster-based architecture, with 20% dynamic work load

5.3 Evaluation of CID-based Server Architectures

In order to evaluate the behavior of real applications running on CIDs, we have developed a detailed event-driven simulator. The simulator is based on the MINT [37] front-end, and allows us to simulate a cluster of SMPs connected to several I/O devices.

We simulate the behavior of an N-way SMP, where each processor has a coalescing write-buffer and a first-level cache. Second-level caches are external to processors and implement a three-phase write-back coherence protocol. The memory hierarchy is completed with a memory controller and M independent memory banks. The processors and caches are connected to the memory through the memory bus. An I/O bridge is also plugged into the memory bus and serves as an interface between the memory bus and the I/O interconnect. Disk controllers and network interfaces can be plugged into the I/O interconnect.

Our simulator is able to implement several architectures for the I/O interconnection. For this paper we evaluate an I/O bus (PCI) and a switch-based I/O interconnect (InfiniBand [18]).

Contention for buses and memories is simulated in detail for all I/O architectures. However, contention for the I/O interconnection is only simulated at the end-points. Given the high interconnection bandwidths we are considering, this assumption should not cause relevant inaccuracies. The main architectural parameters used for each one of these implementations are listed in Table 2.

Parameter	Default Value
Number of host processors	1
Clock freq. of host processor	500 MHz
First-level Cache size	32 KB
Second-level Cache size	256 KB
Main memory size	256 MB
Memory bus frequency	133 MHz
Number of NICs	1
Raw NIC transfer rate	1024 Mb/s
Raw PCI transfer rate	512 MB/s
Infiniband 4-bye message latency	4 μs
Infiniband transfer rate	2 GB/s

Table 2: Key simulation parameters and their default value

The simulator allows us to run several application processes or threads on each processor. These processes share all the resources of the SMP on which they execute. In addition, the simulator provides us with a programmable processor for each I/O device whose architecture is similar to that of a host/memory pair. We have written user-level libraries that implement the file system and TCP/IP

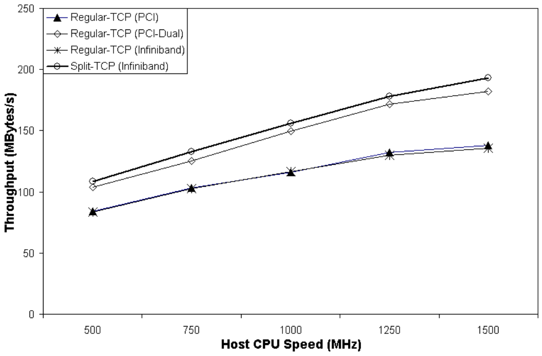

Figure 15: Throughput for various architectures

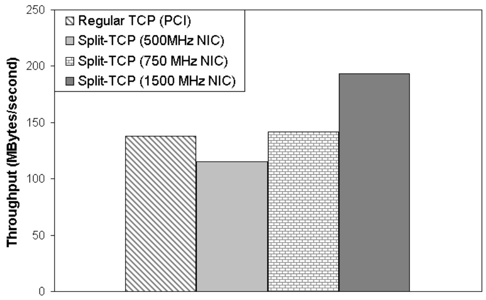

Figure 16: Throughput for various NIC processor speeds

processing. The TCP/IP library is based on the KA9Q [38] TCP/IP package.

For all our simulations, our application is a multi-threaded Web server with a main thread listening to a socket for connections. The thread parses the requests received on each connection and serves out files from the disk or internal cache. The workload consists of repeated requests for 16-KByte files.

In Figure 15, we present the performance of our web server as a function of the host processor speed for four different architectures. Regular-TCP represents the execution of the web server on a uniprocessor system with a PCI bus. Regular-TCP(PCI-Dual) has the web server running on an SMP system with two processors. Regular-TCP(Infiniband) has the TCP server executing on the host processor which is connected to the NIC by a switch-based I/O interconnect like Infiniband. Split-TCP (Infiniband) has the TCP server on an intelligent NIC connected to the host by an I/O interconnect like Infiniband. A message send involves the application sending the message to the NIC. In this architecture, the message is transferred directly from user-space to a buffer on the NIC. All intra-server communication in this architecture is performed with remote memory reads and writes. We can see that for all the simulated processor speeds, the Split-TCP system outperforms all the other implementations. The improvements over a conventional system range from 20% to 45%.

Impact of NIC processor speed: In Figure 16, we present the performance of the web server as a function of the processing power available at the NIC. All these experiments use a 1.5 GHz host processor. These experiments show that the ratio of processing power at the host to that available at the NIC plays an important role in determining the server performance. In the case of Split-TCP (500 MHz), the performance is worse than Regular-TCP (PCI) because the processor on the NIC saturates much earlier than the host processor or the network. We see that we can achieve better performance with a Split-TCP implementation only with a fast processor on the NIC. This seems to validate the decision to use fast processors on commercially available intelligent NICs which can offload TCP/IP processing from the host CPU [20, 11, 40].

6 Related Work

The oldest study about dedicating processors for I/O in a multiprocessor system was done at IBM for the IBM TSS 360 [17]. However, the focus of dedicating the processors, was storage, and networking was not an important design criterion. Since most of the applications were not storage intensive, Amdahl's law [4] was not favourable for such systems.

Intelligent devices have been shown to be a promising innovation for servers, especially in the case of storage systems [16, 1, 9]. Intelligent network interfaces [30] have also been studied, but mostly for cluster interconnects in distributed shared memory [15] or distributed file systems [5]. Recently released network interface cards have been equipped with hardware support to offload the TCP/IP protocol processing from the host [3, 24, 2, 11, 14, 40, 19]. 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. Our work explores the possibility of using dedicated processors for network processing with the advent of intelligent devices capable of executing an entire subsystem, keeping in mind the I/O requirements for the network servers.

OS mechanisms and policies specifically tailored for servers have been proposed in [12, 32, 8]. 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*. This phenomenon was reported by Mogul and Ramakrishna [25]. Several researchers suggest the use of polling on the system to prevent receive livelock and for high performance [36, 23]. Aron and Druschel in [6] use the soft timer mechanism to poll on the network interface. The idea is extended in Piglet [29], where the application is isolated from the asynchronous event handling using a dedicated polling processor in a multiprocessor.

In the Communication Services Platform (CSP) [35] project, the authors suggest a system architecture for scalable cluster-based servers, using dedicated network nodes and a VIA-based network to tunnel the TCP packets inside the cluster. This project has similar goals to our design i.e., offloading the network processing to dedicated nodes. However, their results are very preliminary and their goal was to limit the network processing to a specific layer in a multi-tier data center architecture.

To contrast our work with CSP and Piglet(which is closest) we list our main contributions. 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 intelligent devices. Unlike Piglet or CSP, we study the effect of such separation of functionality for the server systems under real server application workloads.

Recent work [10] was the first effort to study the impact of next generation I/O architectures on the design and performance of network servers. In this work, modeling and simulations were used 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 re-designing the operating system to offload file system and networking functions from the host to intelligent devices.

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 three TCP server architectures: a dedicated network processor in a symmetric multiprocessor (SMP) server, a dedicated node on a cluster-based server built around a memory-mapped communication interconnect, and an intelligent network interface (I-NIC) in a cluster of intelligent devices with a switch-based I/O interconnect.

Based on our experimental results, we can draw a few important conclusions: (1) an SMP-based approach to TCP servers is more efficient than a cluster-based one, given the lower socket channel overheads and the slightly more even division of labor between application and network processors of the former approach, (2) the benefits of SMP and cluster-based TCP servers reach 30% in the scenarios we studied, (3) the simulated results show greater gains of up to 45% for a cluster of intelligent devices, and (4) it is clear that greater gains are possible only if we perform dynamic load balancing between application and network processors.

We are in the process of performing more experiments with each architecture, implementing dynamic load balancing between processors of different classes, and extending our simulation infrastructure to include simulations with device-to-device communication in order to show the advantage of removing the unnecessary buffering at the host.

Acknowledgments

We thank Willy Zwaenepoel for early discussions on splitting the TCP processing on SMP servers.

References

[1] Acharya, A., Uysal, M., and Saltz, J. H. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th ASPLOS* (1998), pp. 81–91.

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

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

[4] Amdahl, G. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Conference* (1967), pp. 483–485.

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

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

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

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

[9] Brown, A., Oppenheimer, D., Keeton, K., Thomas, R., Kubiatowicz, J., and Patterson, D. Istore: Introspective storage for data-intensive network services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)* (March 1999).

[10] Carrera, E. V., Rangarajan, M., Bianchini, R., and Iftode, L. 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).

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

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

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

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

[15] Felten, E. W., Alpert, R. D., Bilas, A., Blumrich, M. A., Clark, D. W., Damianakis, S., Dubnicki, C., Iftode, L., and Li, K. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture* (May 1996).

[16] Gibson, G. A., Nagle, D. F., Amiri, K., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th ASPLOS* (October 1998).

[17] Architecture of the ibm system/360. http://www. research. ibm. com/ journal/ rd/ 441/ amdahl. pdf, 1964.

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

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

[20] Intel(R) IXP1200 Network Processor Family. http://www. intel. com/ design/ network/ products/ npfamily/ ixp1200. htm.

[21] iSCSI: Storage over Internet(IP) Protocol. http://www.iscsistorage.org.

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

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

[24] Lucent Optistar GE1000. http://www.lucent.com.

[25] Mogul, J. C., and Ramakrishnan, K. K. 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* (Berkeley, CA, USA, Jan. 1996), pp. 99–111.

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

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

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

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

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

[31] Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. 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).

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

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

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

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

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

[37] Veenstra, J. E., and Fowler, R. J. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (1994), pp. 201–207.

[38] Wadet, I., and Wade, I. NOSintro: TCP/IP Over Packet Radio; An introduction to the KA9Q Network Operating System. http:// www. netro. co. uk/ nosintro. html, 1992.

[39] Webstone 2.0.1. http://www. mindcraft. com/ webstone/ ws201-descr. html.

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

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