Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O

Tim Brecht, G. (John) Janakiraman, Brian Lynn, Vikram Saletore, Yoshio Turner
Enterprise Software and Systems Laboratory
HP Laboratories Palo Alto

HPL-2005-211(R.1)
April 14, 2006*

network processing, asynchronous I/O, TCP/IP, multi-core processors

Applications requiring high-speed TCP/IP processing can easily saturate a modern server. We and others have previously suggested alleviating this problem in multiprocessor environments by dedicating a subset of the processors to perform network packet processing. The remaining processors perform only application computation, thus eliminating contention between these functions for processor resources. Applications interact with packet processing engines (PPEs) using an asynchronous I/O (AIO) programming interface which bypasses the operating system. A key attraction of this overall approach is that it exploits the architectural trend toward greater thread-level parallelism in future systems based on multi-core processors. In this paper, we conduct a detailed experimental performance analysis comparing this approach to a best-practice configured Linux baseline system.

* Internal Accession Date Only
Published in and presented at EuroSys2006, 18-21 April 2006, Leuven, Belgium
Approved for External Publication

Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O

Tim Brecht[‡] G. (John) Janakiraman[§] Brian Lynn[§] Vikram Saletore[**] Yoshio Turner[§]

[§]Hewlett Packard Laboratories [**] Intel® Corporation [‡]University of Waterloo

ABSTRACT

Applications requiring high-speed TCP/IP processing can easily saturate a modern server. We and others have previously suggested alleviating this problem in multiprocessor environments by dedicating a subset of the processors to perform network packet processing. The remaining processors perform only application computation, thus eliminating contention between these functions for processor resources. Applications interact with packet processing engines (PPEs) using an asynchronous I/O (AIO) programming interface which bypasses the operating system. A key attraction of this overall approach is that it exploits the architectural trend toward greater thread-level parallelism in future systems based on multi-core processors. In this paper, we conduct a detailed experimental performance analysis comparing this approach to a best-practice configured Linux baseline system.

We have built a prototype system implementing this architecture, ETA+AIO (Embedded Transport Acceleration with Asynchronous I/O), and ported a high-performance web-server to the AIO interface. Although the prototype uses modern single-core CPUs instead of future multi-core CPUs, an analysis of its performance can reveal important properties of this approach. Our experiments show that the ETA+AIO prototype has a modest advantage over the baseline Linux system in packet processing efficiency, consuming fewer CPU cycles to sustain the same throughput. This efficiency advantage enables the ETA+AIO prototype to achieve higher peak throughput than the baseline system, but only for workloads where the mix of packet processing and application processing approximately matches the allocation of CPUs in the ETA+AIO system thereby enabling high utilization of all the CPUs. Detailed analysis shows that the efficiency advantage of the ETA+AIO prototype, which uses one PPE CPU, comes from avoiding multiprocessing overheads in packet processing, lower overhead of our AIO interface compared to standard sockets, and reduced cache misses due to processor partitioning.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management—*Network communication*; C.4 [**Performance of Systems**]: Design Studies; C.5.5 [**Computer System Implementation**]: Servers

General Terms

Performance, Design, Experimentation

Keywords

Network Processing, Asynchronous I/O, TCP/IP

1. INTRODUCTION

Network protocol processing can saturate servers when running high-speed TCP/IP-based applications. High overhead for packet processing can leave few CPU cycles available for actual application processing. This problem is worsening as servers deploy higher bandwidth 10 GigE interfaces. One solution that has been proposed is to offload TCP/IP processing from the host CPUs to the network interface to accelerate network processing and free host cycles for application processing. However, prior TCP offload investigations [9, 46, 47, 33, 21] have not demonstrated such improvements and point to the limited capabilities of the TCP Offload Engine (TOE) hardware. While future TOE designs will likely show improvements, the processing elements in TOEs are likely to always be behind the performance curve of mainstream processors. Furthermore, memory limitations on TOE devices can force them to rely on host support in high load scenarios, degrading performance.

Several evolutionary trends in server hardware and networking software stacks have renewed focus on improving the packet processing efficiency of servers without resorting to wholesale TCP offload [50, 42, 43, 41, 12]. The higher degree of parallelism enabled by multi-core, multi-threaded processors is a good fit for concurrently processing multiple network streams. Increasing cache sizes and shared caches on multi-core chips provide greater potential to hide memory latency by transferring data between processors through the caches (e.g., between the application and the network stack). Memory bandwidth and I/O bandwidth will scale with the number of processing elements as server designs transition to high-bandwidth point-to-point links (from multi-drop shared buses) for memory and I/O [11]. Furthermore, memory controller interfaces and network interfaces will be integrated much closer to processor cores reducing memory and I/O overheads and bottlenecks. Concurrent with these hardware developments, the socket/TCP/IP network stack is evolving [4] to adopt concepts such as OS-bypass and asynchronous I/O from APIs such as VIA [17] and InfiniBand [26] to reduce operating system overhead and increase I/O concurrency. Our research is aimed at leveraging these trends to support scalable network performance on standard server platforms.

In this direction, we and others [50, 42, 43, 41] have suggested that network packet processing efficiency can be improved by dedicating a subset of the server's processors for network processing and by using asynchronous I/O for communication between the network processing engine and the application. In our architecture, the **Embedded Transport Acceleration (ETA)** [42, 50] component (developed at Intel® Labs) dedicates ("sequesters") one or more processors or hardware threads in the server to perform all network processing. These dedicated Packet Processing Engines (PPEs) avoid context switches and cache conflicts with application processes, and they avoid costly interrupts by polling applications and network interfaces. PPEs can poll efficiently since they are fully integrated in the system's cache coherence domain. ETA also exposes VIA-style [17] user-level communication to applications, reducing data copies and bypassing the operating system for most I/O operations [26, 3]. The second component of our architecture exposes **Asynchronous I/O (AIO)** based socket semantics to applications [7, 6, 4, 23]. The APIs support asynchronous versions of traditional socket operations. Applications issue operations without being blocked and without first verifying socket descriptor status. Applications receive asynchronous *events* that signify the completion of previously issued I/O operations. The AIO model allows applications to sustain concurrent I/O operations without the overhead of multiple software threads. The queues of operations and events between the application and the packet processing allow these two functions to independently speed up or slow down with respect to each other, enabling greater concurrency between them. The combined ETA+AIO architecture enables the development of highly concurrent, low overhead, event-based networking applications. Event-based applications are structured as state machines in which a state transition occurs when the application processes an event (e.g., when the OS or an I/O device notifies the application that an outstanding operation has completed) [51].

This paper makes two primary contributions. First, it provides the first rigorous analysis of this approach based on processor partitioning and asynchronous I/O, and identifies its strengths, weaknesses, and opportunities. We have ported an open source web server, userver [14], to a prototype implementation of the ETA+AIO architecture and compared its performance against performance on a baseline Linux system configured using best practices. Our experimental results show that processor partitioning combined with AIO can improve network processing efficiencies over the best Linux configuration, but these benefits are small in magnitude for the prototype system. Our analysis shows that the efficiency advantages of the prototype system result primarily from instruction path length improvements through the use of AIO and a polling architecture, and to a small degree from reduced cache misses. The AIO API allows the application to interface with the PPE directly through shared memory, bypassing operating system overheads associated with issuing socket operations and receiving event notifications. ETA's polling-based dedicated PPE architecture avoids interrupt related overheads, for example by substituting softirq mechanisms with lighter-weight in-line function calls, and it results in reduced cache misses by eliminating cache conflicts between the application code and the network stack.

Our second main contribution is to provide a detailed description of the changes required and issues that arose when modifying the event-driven userver to take full advantage of the asynchronous socket I/O model. Our experience with extending the userver to use the AIO API illustrates several new aspects that should be factored in when developing AIO-based applications. In particular, the use of an AIO model necessitates additional steps for registering and deregistering (and, therefore, managing) buffer memory, simplifies the application by eliminating the need to handle partially completed operations, exposes new considerations in the sequencing of the application's socket operations, and enables applications to process events and issue operations using application-specific policies to improve resource management and performance. In addition, given the similarity of our AIO API to the Sockets Extension API [4] being developed in the industry, our implementation of AIO over ETA is potentially useful as an example implementation for the Sockets Extension API.

The remainder of this paper is organized as follows. We discuss related work in Section 2. Section 3 provides an overview of the ETA+AIO architecture, followed by a detailed discussion of the ETA implementation in Section 4 and the AIO API implementation in Section 5. Section 6 describes the architecture of our userver application and its modification to use the AIO API. Section 7 discusses the results from our performance evaluation of ETA+AIO. Potential architectural extensions are discussed in Section 8, and we conclude in Section 9.

2. BACKGROUND AND RELATED WORK

A network server application can be implemented using a *multithreading* programming model, an *event-driven* programming model, or a combination thereof [39]. The ETA+AIO architecture is well suited to the more efficient event-driven model, which is free of the thread scheduling and context switching overheads inherent to multithreading. With the event-driven model an application consists of a single execution thread per CPU. To avoid pausing the single thread for extended durations, the thread must use either *non-blocking* or *asynchronous* I/O (AIO) to perform I/O operations. The standard network sockets interface supports non-blocking I/O operations which execute synchronously with the application and return prematurely if they cannot complete without blocking [49]. Applications use an *event notification* mechanism, such as Unix `select` or `poll` or Linux `epoll` [32, 38], to detect changes in socket state that indicate which non-blocking operations could make progress if issued.

In contrast to non-blocking I/O, with AIO an application issues non-blocking calls which post operations that execute to completion asynchronously to the application. When the operations eventually complete, the application is notified, usually through an event queue mechanism. AIO thus eliminates the partially completed operations which can burden applications that use non-blocking I/O. There is a growing trend toward enabling applications to use AIO in addition to non-blocking I/O. For example, Lazy AIO has recently been proposed as a general OS mechanism for automatically converting any system call that blocks into an asynchronous call [18]. With Lazy AIO, calls that do not have to block execute to completion synchronously to the application (similarly to non-blocking I/O), whereas calls that would have to block are transformed automatically into asynchronous operations, which require completion event generation. The more specialized POSIX AIO API supports a set of asynchronous read and write operations and uses POSIX real-time signals to provide an event queue for notifying applications of operation completion [7]. Microsoft Windows provides extensive support for AIO, with notification provided through OS event queues called completion ports [23]. The Open Group, which defines Unix standards, has recently approved the Extended Sockets API, which provides extensive AIO capabilities for network sockets [4]. Extended Sockets differs from prior asynchronous sockets mechanisms by providing OS bypass primitives inspired by previous work such as VIA [17]. Specifically, primitives for memory registration and for shared memory work queues and event queues enable applications to post operations and receive completion notification without going through the OS kernel. In this paper, we use an AIO API that provides socket-like primitives and OS bypass capabilities very similar to Extended Sockets. In Section 6 we present our experiences with using this API to implement an open source web server application.

To support applications, the underlying system performs TCP/IP packet processing. The high cost of this processing has been a long-standing concern. In 1989, Clark et al [15] argued that the TCP protocol itself can be implemented efficiently in software, but operations like buffer management, connection structure lookup, and interrupt handling, which would all be needed with any transport layer protocol are often expensive and limit performance. Kay and Pasquale [29] later reported the high cost of data touching operations (e.g., copies and checksum computations), and Mogul and Ramakrishnan [34] showed that under high traffic loads interrupt processing could easily overwhelm a server. These and related studies [30, 20, 19] have proposed various techniques to improve packet processing efficiency.

Recent OS kernels provide features that can reduce network processing overhead. The "zero-copy" `sendfile` system call enables data transfer between file descriptors without requiring the data to be copied through user space buffers. Processor affinity for interrupts reduces device driver contention for network hardware and kernel data structures [19]. Software-based interrupt moderation (e.g., NAPI [45]) can help reduce the overhead of interrupt processing under high loads. Many of these features are not useful in the ETA+AIO architecture, which bypasses the kernel for most data transfer operations. However, the ETA+AIO architecture provides similar benefits, avoiding copies on the transmit path like `sendfile`, and eliminating the overhead of network device interrupts by using a polling interface.

To reduce heavy loads on host CPUs, recent network interface card (NIC) designs provide some degree of *offloading*, the ability to execute some packet processing operations that previously had to execute on the CPU. In particular, modern mainstream NICs (e.g., Intel$^{®}$ PRO/1000 gigabit Ethernet) can perform TCP/IP checksums and segmentation, and provide programmable timers used for interrupt moderation [27]. Because offloading changes the software interface to the NIC, OS device drivers and the protocol stack software must be designed or modified to take advantage of these capabilities. The ETA architecture can benefit as much as conventional systems from NIC offloading capabilities since the ETA architecture makes only slight modifications to a standard TCP/IP stack implementation.

Although offloading selected functions has proved successful, so far there is only limited use of TCP Offload Engine (TOE) NICs, which offload the entire TCP/IP processing for some or all packets to the NIC. TOEs typically rely on firmware and specialized hardware implementations, which are more difficult to upgrade and customize than host software-based implementations. TOEs have not yet demonstrated significant performance benefits [9, 46, 47, 33], possibly because the current scarcity of NIC hardware resources limits scalability, rendering a TOE better suited for supporting a few long-lived TCP connections than for workloads with a large number of short-lived connections (e.g., web server workloads) [33]. In recent work, Freimuth et al [21] propose a TOE design to minimize bus crossings, data copies, cache misses, and interrupts, which they argue limit the scalability of current systems. Using a purely software-based emulation environment, they demonstrate that their TOE design reduces I/O bus utilization (they plan to evaluate other performance aspects in future work). Hence, they cannot yet assess the overall performance benefits of this approach. In this paper, we take a different evaluation approach. We evaluate a real multiprocessor server implementation of the ETA+AIO architecture and measure network throughput and CPU usage. We present a detailed analysis of these results which quantifies the impact of different elements of the ETA+AIO architecture on microarchitecture performance attributes including instruction execution rates, cache misses, and interrupt processing overhead.

At the microprocessor level, the growing challenges of power density and interconnect delay are limiting further increases in single-core processor performance. Thus, the emerging design trend is to use increasing transistor density to increase the integration of functions on a chip and to implement multi-core processors. As multi-core processors decrease the relative cost of individual cores and provide mechanisms for fast inter-core communication, it becomes more attractive to dedicate some cores to packet processing, as in the ETA+AIO architecture. In addition, this approach is complementary to proposals for placing network interface hardware close to or even integrated onto the processor chip [12], and techniques for intelligently transferring data directly between network hardware and CPU caches [12, 25].

Previous studies of an earlier version of our ETA+AIO prototype indicate that it improves performance for simple microbenchmarks [42, 44]. In contrast, our evaluation vehicle is a full-fledged event-driven web server application driven by a SPECweb99-like workload [48]. The web server application makes full use of `sendfile`, non-blocking I/O and `epoll`, and an aggressive connection acceptance strategy.

An earlier (1998) approach that also dedicated CPUs to device processing was the AsyMOS [36] and Piglet [37] project. The mechanisms proposed in that project appear to provide more generality than ETA, allowing for example the ability to download user code to the dedicated device processors. However, the published evaluations include only device driver code on the device processors, while TCP is left for the application CPUs. In addition, only a very preliminary performance analysis is presented, and of course the analysis is based on the relatively unsophisticated hardware and OS features of that time period.

TCP Servers [41], a more recent project similar to ETA, has examined the impact of sequestering CPUs for dedicated TCP processing. The TCP Servers project provides an AIO API that supports zero-copy data transfers [40], but this AIO API is implemented only for transfers between different nodes of a cluster interconnected by a system-area network. On a multiprocessor platform, the implementation of TCP Servers provides neither zero-copy transfers nor AIO for interactions between applications and the dedicated packet processing CPUs. In contrast, our prototype provides these features in a multiprocessor implementation. The TCP Servers study finds that offloading TCP/IP processing can improve server throughput by up to 30%, that noticeable benefits are obtained by using network interface polling rather than interrupts, and that substantial computing resources are required for complete offloading. In addition, the authors note that server performance would further benefit from a scheme for dynamically adjusting the allocation of CPUs for application and network processing.

Unlike both the TCP Servers work and the previous ETA+AIO investigations, in this paper we explicitly ensure that we compare the ETA+AIO prototype against a *best-practice configured* standard Linux system with interrupt and application processor affinity, network interface interrupt moderation, and NIC checksum offloading. Our findings are significantly different from these previous studies. We observe smaller benefits from partitioning since we compare against a strong baseline system that benefits, for example, from transmit-side copy-avoidance and interrupt moderation. We characterize the remaining benefits of the ETA+AIO architecture, which include higher efficiency of the AIO API compared to sockets and improved caching behavior.

Finally, Jacobson and Felderman [28] very recently proposed techniques for speeding up network processing by using lock-free circular buffers in the network stack instead of traditional linked lists of socket buffer structures, and also by performing TCP processing in a user-level library linked with the application. Since ETA uses the standard TCP stack, the benefits of changing the stack implementation to use circular buffers would likely accrue to both ETA and the standard system. However, running the stack in a user-level library on the same CPU as the application is contrary to ETA's use of processor partitioning, and further study is needed to evaluate the tradeoffs between these opposing approaches. For example, while partitioning reduces contention for CPU resources between network processing and application processing, some functions (e.g., copying received packet data payloads from kernel memory to user-level memory) are probably more cache-friendly when performed on the same CPU as the application.

3. ARCHITECTURE OVERVIEW

Our architecture dedicates one or more processors or hardware threads of a server to perform all TCP/IP protocol processing. Applications interface to the ETA TCP/IP stack through an API called the Direct User Sockets Interface (DUSI). DUSI provides an asynchronous interface similar to the Sockets Extension API [4]. It presents familiar socket semantics to applications with asynchronous versions of listen, accept, send, receive, etc.

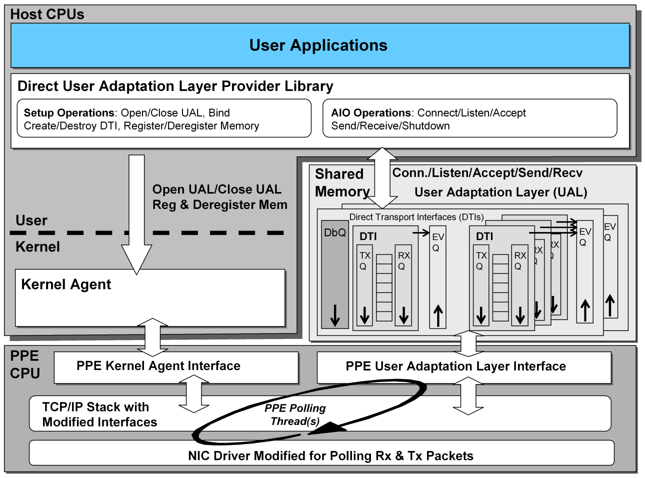

Host CPUs
User Applications
Direct User Adaptation Layer Provider Library
Setup Operations: Open/Close UAL, Bind Create/Destroy DTI, Register/Deregister Memory
AIO Operations: Connect/Listen/Accept Send/Receive/Shutdown
Open UAL/Close UAL Reg & Deregister Mem
User
Kernel
Kernel Agent
Shared Memory
Conn./Listen/Accept/Send/Recv
User Adaptation Layer (UAL)
Direct Transport Interfaces (DTIs)
DbQ
DTI
TX Q
RX Q
EV Q
PPE CPU
PPE Kernel Agent Interface
PPE User Adaptation Layer Interface
TCP/IP Stack with Modified Interfaces
PPE Polling Thread(s)
NIC Driver Modified for Polling Rx & Tx Packets

Figure 1: DUSI/ETA PPE Software Architecture

Figure 1 depicts the architectural components and partitioning between the "sequestered" CPUs and the "host" CPUs. The Packet Processing Engine (PPE) is a kernel thread running on the sequestered CPU, and takes no part in user applications. The Direct User Adaptation Layer Provider Library (DUSI Library) implements the AIO functions and is linked with applications executing on the host CPUs. Applications and the PPE interact through queues created in cache coherent shared system memory. Queuing and dequeing operations and events are extremely fast lock-free operations requiring only a few instructions and no kernel intervention. Finally, a Kernel Agent (KA) is used for those operations for which the PPE requires Linux kernel assistance (for example, pinning data buffers in memory).

Applications establish communication with the ETA PPE by creating a User Adaptation Layer (UAL). The UAL houses all queuing structures shared between an application and the PPE. To initiate TCP/IP communication, an application creates a Direct Transport Interface (DTI) within the UAL, analogous to creating a traditional BSD socket. A DTI contains or references all queuing structures associated with a TCP socket [50, 44]. The UAL and DTIs are created using synchronous operations. Subsequently, the application can issue socket operations through the DTI queues and receive completion events asynchronously.

The PPE executes a polling loop that continuously queries NIC descriptors and ETA queues. Since all NIC descriptors and ETA queues are accessed through shared cache-coherent system memory, polling is performed directly from cache and has no effect on the memory subsystem or front-side bus. A memory access occurs only when a memory location has been updated by either a NIC or the DUSI Library [42]. Another benefit of polling is that the PPE avoids incurring interrupts. The PPE uses a modified Ethernet driver that allows it to perform NIC polling, and it incorporates a TCP/IP stack that is a slightly modified version of the mainstream Linux TCP/IP stack.

4. ETA ARCHITECTURE

In the following subsections we describe the queue structures used in the Embedded Transport Acceleration (ETA) architecture and the operation of the PPE thread.

4.1 ETA Queues

Each application has one instance of a UAL which contains (see Figure 1) one Doorbell Queue (DbQ), one or more Event Queues (EvQs), and, for each DTI, a pair of Transmit and Receive descriptor queues (TX and RX). Memory for these queuing structures is pinned and mapped into both the user and kernel virtual address spaces to permit access by both the DUSI Library and the PPE. The DUSI Library posts to the Doorbell Queue to alert the PPE whenever the application initiates an operation on any of its DTIs. This enables the PPE to discover application-initiated operations by polling only the DbQ instead of polling each of the queues of several DTIs. The application creates Event Queues in the UAL for use by the PPE to post events notifying the application of the completion of previously issued operations. To wait for events, applications can either poll the EvQs or request notification through the OS signal mechanism.

A DTI is associated with each instance of a socket (whether it is an unbound or listening socket, or the endpoint of a TCP stream). When an application initiates a send (receive) operation on a DTI, pointers to the application's send (read) buffers are posted by the DUSI Library to the DTI's TX (RX) descriptor queue (the DUSI Library also posts to the DbQ as mentioned above). When an application creates a DTI, it can bind the TX and RX queues of the DTI to any EvQ within the UAL. This flexibility allows the application to combine event notifications for operations associated with several DTIs into a smaller number of EvQs, reducing the number of EvQs it must poll.

In addition to the UAL queues, ETA includes the Administrative Queue (AQ) and the Interrupt Queue (IQ) through which the Kernel Agent and the PPE communicate. To perform synchronous operations, the DUSI library invokes an `ioctl` system call to schedule the Kernel Agent, which then queues a request to the AQ. The PPE reads requests from the AQ and uses the IQ to notify the Kernel Agent of the completion of AQ requests and to trigger the Kernel Agent to deliver signals requested by the application. The PPE posts an event to the IQ and then generates an Interprocessor Interrupt (IPI) for the host CPU to cause the Kernel Agent to be scheduled.

4.2 PPE Polling Loop

The PPE's goals include ensuring that: NICs do not drop packets, NICs are never idle when there is data to be transmitted, and latency is minimized for all application operations. This requires a balance between how aggressively the PPE takes on new operations posted by applications and how often it services NICs.

Our prototype PPE uses the algorithm shown in Figure 2. Within this loop the TCP/IP stack can be invoked in several places. The PPE contains a Linux 2.4.16 TCP/IP protocol stack that includes minor modifications for it to interface to the PPE queuing mechanisms. Whenever the PPE thread finds that a NIC has received a packet (Step 4), it invokes the inbound TCP/IP stack to service the packet. When the PPE dequeues an application's send operation from the DbQ (Step 3), it invokes the outbound TCP/IP stack to generate and send packets to the network. The PPE can also invoke the outbound TCP/IP stack to resume a blocked transmis-

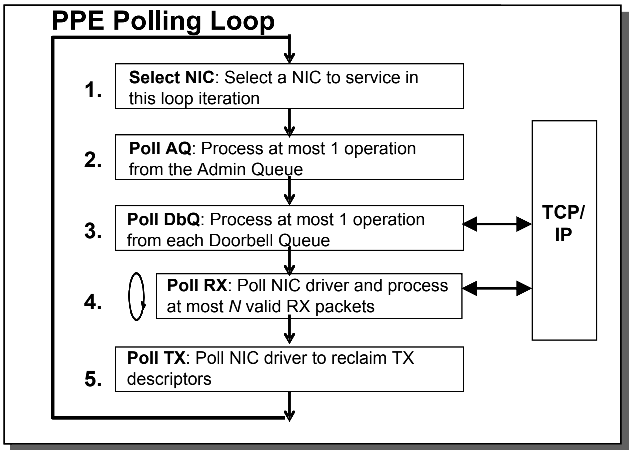

PPE Polling Loop
1.
Select NIC: Select a NIC to service in this loop iteration
2.
Poll AQ: Process at most 1 operation from the Admin Queue
3.
Poll DbQ: Process at most 1 operation from each Doorbell Queue
4.
Poll RX: Poll NIC driver and process at most N valid RX packets
5.
Poll TX: Poll NIC driver to reclaim TX descriptors
TCP/IP

Figure 2: PPE Polling Loop

sion that was initiated in some previous execution of Step 3. This occurs in Step 4 when an ACK is received that opens the TCP window, allowing the blocked transmission to resume. A simple scheduling policy that we have found effective is to poll the NICs in a round-robin fashion, one NIC per loop iteration, and to dequeue at most one operation from each DbQ in each iteration.

When using a single NIC and running one DUSI application on a 2.8GHz Xeon™ server, an unproductive iteration through our prototype's PPE loop – where all queues are empty and the NIC has no packets – on average takes only 152 clock cycles (∼55ns). We have found that when the PPE has ample CPU cycles available to handle the load, the choice of scheduling policy has almost no effect on performance; the PPE achieves a fair balance between application and NIC processing since the queues tend not to have persistent backlogs and the PPE is work-conserving (the cost of polling an empty DbQ or an empty NIC is minimal). Only when the PPE starts to saturate the CPU does it matter how effectively the scheduling policy allocates PPE CPU cycles for polling the NICs versus polling the application DbQs. In this operating region, queues are longer and should be drained at appropriate rates. Allocating too few cycles for handling DbQ entries would cause the DbQs to become unacceptably long, delaying the processing of application requests. Conversely, devoting more cycles to handling DbQ entries would leave fewer cycles for processing network traffic. Thus, it may be beneficial to dynamically adjust resource allocations to optimize the handling of different workloads.

5. ASYNCHRONOUS I/O (AIO)

Our prototype implements an asynchronous interface, DUSI, which allows applications to execute unimpeded while networking operations are being performed. Before a DUSI application can send and receive data over TCP/IP, it must first call the synchronous function `Dusi_Open_Ual` to create a User Adaptation Layer and initialize application-specific data structures. In creating a UAL, an application must specify the depth of the Doorbell Queue and the maximum number of DTIs that will be created. As with all synchronous DUSI functions, creating a UAL interacts with the ETA Kernel Agent thread rather than directly with the PPE. This provides access to kernel functions, such as for pinning memory and translating user virtual addresses to kernel virtual addresses, and allows slower operations to be offloaded from the PPE.

An application is responsible for creating the Event Queues on which it receives notification of completed operations. When an application creates a DTI socket, it associates two EvQs with the DTI: a receive EvQ (RX EvQ) and a transmission EvQ (TX EvQ). The RX EvQ is used by the application for receiving notification of when inbound operations have completed (e.g., receives, accepts). The TX EvQ is associated with outbound operations (e.g., send, shutdowns, application initiated connections). DUSI does not put constraints on which event queues may be associated with DTIs. For example, it is permissible for a DTI to use a single EvQ as both the RX and TX EvQs; and EvQs may be shared amongst multiple DTIs.

To establish a TCP connection an application must first create the DTI socket to be associated with the stream. If the application is the listener, then the application must also create the listen DTI socket and bind it to a TCP port address. In contrast to BSD sockets, where an `accept` operation returns the TCP stream socket descriptor upon completion, a DUSI application first creates the DTI stream (child) socket and then passes it, along with the DTI listen socket, as input parameters to the `Dusi_Accept` function. This permits the application to initiate a receive on the TCP stream before an accept operation has been posted. The sequence ensures that an application data buffer will be available to the PPE for any data immediately following the TCP SYN.

To send data an application calls `Dusi_Send` with a pointer to the source data and the data's length. The DUSI library writes the information to the DTI's TX Descriptor Queue, and then notifies the PPE by posting to the DbQ. Similarly, the application initiates a read by calling `Dusi_Recv`, which results in an event being queued to an RxQ followed by the PPE being notified via the DbQ. Vectored sends and receives, specifying multiple data locations, are supported. By writing to an asynchronous interface an application can have multiple send and receive operations outstanding for a single TCP stream, as well as across multiple TCP streams. This can lead to higher network bandwidth and less latency for a single application.

Since an application executes in user space and the PPE executes as a kernel thread, they do not share virtual addresses. When an application references a buffer, the application's virtual address needs to be mapped to a virtual address accessible by the PPE. To perform this mapping we require an application to register memory prior to referencing it in send and receive operations. Registering memory returns a handle to be included in calls to DUSI I/O functions. When memory is registered ETA makes certain that all user memory, including anonymous memory, is backed by physical pages. ETA also takes advantage of registration to pin the pages in memory, preparing the pages to be accessed by the NIC.

Prior research [22, 31, 16] has shown the benefits of copy avoidance in network performance. When sending data, ETA minimizes copies by writing directly from the user buffer to the NIC. This is easily and efficiently done because the application buffer is already mapped to a kernel virtual address and pinned in memory. Since retransmissions may need to occur from that application buffer, ETA does not generate a send completion event until the remote TCP has acknowledged all of the packets. An AIO interface lends itself nicely to sending from an application buffer since the application is never impeded on the send and, in contrast to BSD sockets, there is never a need to indicate partial success.

For inbound data, the PPE must copy data from the host NIC buffers into the application's buffer. If a receive is posted at the time that the PPE polls the NIC, then the PPE is able to immediately copy the data to the application's buffer, which can result in reduced latency for receives. We allow an application to have multiple receives posted to a DTI so that packets can be passed to the application as soon as the PPE detects them.

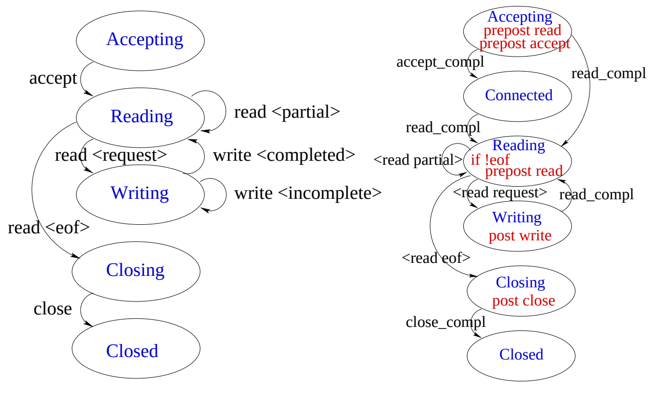
DUSI provides several mechanisms for applications to determine when an operation has completed. There are very efficient functions for polling EvQs, dequeuing events, and determining the EvQ event depth. At times, an application may have operations outstanding but can do no further work until one of the operations is completed. DUSI provides functions that allow an application to block on a single queue, but DUSI currently relies on the Linux user signal mechanism to block on multiple EvQs. A function is provided for associating a signal with an EvQ (the same signal can be assigned to multiple EvQs). If the application wants to sleep until an event arrives, it arms the corresponding signal(s) and notifies DUSI to raise a signal when any of the specified EvQs transitions from being empty to not empty.

6. WEB SERVER APPLICATION

To investigate the benefits of using ETA+AIO for network intensive workloads, we have adapted a web server application to use the DUSI AIO API. A web server is a widely used and representative example of a networking application that exerts high demands on system resources as a result of handling large numbers of simultaneous connections.

Nearly all network server applications follow a similar series of steps to handle each client connection. The server receives a client request on the connection, processes the request, and sends a reply. This continues until the server replies to the final client request and then closes the connection. Several of these steps can block on interaction with a remote host, the network, some other subsystem (e.g., a database), and potentially a disk. Consequently, for high performance the server must handle up to several thousand simultaneous connections [10] and quickly multiplex connections that are ready to be serviced. One approach is to assign each connection to a separate thread of execution which can block [1, 39]. However, the cost of frequently switching execution contexts can limit the application's performance.

An alternative approach which avoids this problem is to use a Single Process Event Driven (SPED) application architecture (terminology from [39]). A SPED architecture uses a single process (per CPU) running an infinite event processing loop. In each loop iteration an event notification mechanism identifies a connection that is ready for additional processing, and then the processing for this connection is performed in a non-blocking fashion [13, 5]. With this approach, a process can service a large number of connections concurrently without being blocked on an I/O operation and without the overheads of multithreading [52]. However, it may still be necessary to use multiple processes per proces-



(a) Baseline (b) AIO

Figure 3: State machine for handling a single connection

sor to mask unavoidable blocking (e.g., for disk I/O on a page fault [39]).

The userver [14, 24] is an open source web server implemented with the SPED architecture, originally using APIs provided by standard Linux. We have extended the userver implementation with the option to use AIO APIs. The following subsections describe the primary aspects of the userver implementation for these two environments. We focus on how the userver handles requests for static content (file data) as opposed to dynamic content.

6.1 The userver for standard Linux

To realize the SPED model on standard Linux, the userver performs socket I/O operations in non-blocking mode as provided by the traditional BSD sockets interface. The userver provides an option to transmit web page data using the `sendfile` system call. Linux provides a "zero-copy" `sendfile` implementation by transmitting data directly from the shared file system buffer cache in kernel memory to the NIC, thus eliminating the copy from user memory to kernel memory incurred with basic socket send operations. The use of `sendfile` improves performance in the baseline system and allows us to investigate the benefits of the ETA architecture aside from its elimination of memory-to-memory copying on the transmit path. Figure 3(a) shows a state machine diagram illustrating how the userver handles a single connection in the baseline system.

6.2 The userver with AIO: Overview

We next describe in some detail our extensions to the userver to use an AIO API. Our description serves as a case study illustrating the use of AIO for a real application. To enable the possible use of a variety of AIO APIs, we modified the userver to use an application-specific AIO API. We then implemented a library that maps the application-specific API to DUSI. The application-specific API is a relatively straightforward adaptation of standard socket calls except that they are asynchronous. For example, some of the calls it supports are `aio_accept`, `aio_read`, `aio_write`, and `aio_close`. Since this mapping is straightforward we do not elaborate on it here but instead focus on how the userver, through the mapping layer, ultimately interacts with the DUSI API.

6.3 The userver with AIO: Startup

At startup the userver calls DUSI to create the user adaptation layer (UAL), EvQs, and multiple child sockets, and to create and bind listen sockets. The userver also registers buffer memory for receiving client HTTP requests. Before entering its event processing loop, the userver posts an accept operation for each child socket to the listen socket's DTI RX descriptor queue in anticipation of client connection attempts. As discussed in Section 5, the userver is able to post a read operation to the child socket before posting the accept operation, ensuring immediate delivery of the client's initial HTTP request by the PPE*.

6.4 The userver with AIO: Event Processing

To use AIO with the SPED architecture, the userver executes an infinite event processing loop where each iteration performs the following steps: 1) dequeue completion events, 2) use the event information provided in the completion event to determine what processing will be done next and on which connection, and 3) issue the appropriate AIO operations on the connection. Figure 3(b) shows a state machine diagram illustrating how the userver responds to events for a single connection.

The userver provides runtime options to define the number of EvQs used and to select the policy for servicing the EvQs. For our experiments we chose to use three EvQs: one for all accept completions, one for receive completions across all DTIs, and one for send and shutdown completions across all DTIs. Using this segmentation enables the userver event processing loop to give highest priority to dequeuing receive completion events. This is beneficial because, as illustrated in Figure 3(b), it is primarily the receive completions that drive the userver's state machine, and other events can often be ignored. For example, the first receive completion dequeued for a DTI can be used as an implicit accept, rendering accept completions superfluous. When the userver does dequeue an accept completion, it can be discarded.

When the userver dequeues a successful receive completion event, then (for HTTP 1.1 connections) the userver immediately posts a new receive operation on the connection's RX descriptor queue in anticipation of receiving the next request from the client. The userver parses the current HTTP request and generates a reply which consists of an HTTP header and the reply data (in the case of a static request, the contents of the file). To enable the PPE to transmit the requested file data directly from memory, the userver `mmaps` requested files and registers the `mmapped` memory regions with DUSI. As part of memory registration, DUSI invokes the kernel agent to pin the memory region into physical memory until it is later unregistered by the application. The userver provides a further optimization by maintaining a cache of recently requested `mmapped` file handles and addresses, along with a cache of corresponding HTTP response headers†. Registering an `mmapped` file

*Our current implementation relies on the simplifying assumption that each HTTP request fits in one packet and therefore posts only a single receive at a time.

†The ETA+AIO prototype does not currently coalesce data in a vectored `Dusi_Send` call. Furthermore the prototype does not implement `getsockopt` and `setsockopt` functionality, most notably `TCP_CORK` and `TCP_UNCORK`. As a result there is currently no mechanism to force the coalescence of an HTTP header and HTML data into a single network

can cause the application to block because of disk accesses as file data is brought into memory. To avoid this blocking, we would need to extend the ETA+AIO implementation by adding a new, asynchronous version of the DUSI memory registration operation. To send a reply, the userver posts to the connection's TX descriptor queue a send operation that includes a list of pointers to the HTTP response header and the requested file data in the registered memory region. The PPE generates a completion event for this send operation once all of the data is ACKed by the TCP peer. Similar to accept completions, send completions (for non-pipelined HTTP requests) are largely superfluous because subsequent receive completions on a DTI can double as implicit send completion events. (However, send completions are useful for managing the userver file cache use count.)

When using HTTP 1.1, a client sends a FIN on a connection to indicate it will not send any more requests to the server. After the server's PPE receives the FIN, any outstanding or subsequent userver receives will be successfully completed with a length of zero. The userver reacts to a zero-byte receive completion event by posting a shutdown operation to the connection's TX descriptor queue. The PPE processes the shutdown by trying to complete pending transmits for the connection and then sending a FIN to the client. When the shutdown is complete, the PPE queues a shutdown completion event to the userver. The userver responds by posting an accept to the listening socket's DTI RX descriptor queue to allow the server to accept a new connection, thus replacing the connection that was just closed. As described earlier, the userver also posts a receive operation to the child DTI. As a result, the sum of the number of active connections plus the number of new connections that can be accepted remains constant. This is similar to a listen queue in the case of standard BSD sockets.

6.5 The userver with AIO: Summary

In adapting the userver to use asynchronous I/O, we identified several important considerations. One is that some events can implicitly indicate the completion of other events. For example, the successful reception of data could be an indication that an accept or prior send has also completed. With this in mind, different event types can be assigned to separate queues, thus permitting the userver to make application-driven decisions about which events to prioritize. Another important factor is having the AIO interface allow reads to be posted on future connections before they are accepted. This can be used to avoid race conditions where the application is notified of a connection being accepted but is unable to post a read request prior to the data arriving. This is important because if a read is posted the PPE can immediately deliver a notification to the application, otherwise it must delay delivering the event until the read is eventually posted. Finally, an asynchronous I/O interface can be very efficient. However, if there are no events to be processed the application must wait for events, and kernel assistance is required to block the application and later unblock it when an event does arrive. As we discuss in Section 7.3 the overhead of using signals as an event notification system can be costly relative to other costs associated with AIO.

packet. To avoid this artifact in our analysis, we embedded the HTTP header into the URI files. This has a very small positive effect on the baseline experiments in that the header is sent via `sendfile` rather than from a user buffer.

7. PERFORMANCE EVALUATION

In this section, we compare the performance of the userver application on the ETA+AIO prototype system against its performance on well-configured standard Linux systems. We also analyze how various aspects of the ETA+AIO architecture affect its network processing efficiency.

7.1 Experimental Environment

Our experiments study the performance of the userver running on a multiprocessor server, under a SPECweb99-like workload generated by a cluster of client machines running `httperf` [35]. To stress computational and networking capabilities on the server, we choose a workload that minimizes disk I/O. Clients issue requests to static files with no think time. We use a small SPECweb99 fileset (51 MB, ten directories) that fits entirely in memory and is completely pre-loaded into memory by the userver at startup. For ETA+AIO the userver also registers the mapped files with the PPE, causing them to be pinned in memory. It is important to note that with this workload choice, our results for the ETA+AIO system characterize its performance behavior without factoring in the cost of memory registration or deregistration operations, which could be significant sources of overhead for other workloads.

The server hardware platform is an HP Proliant DL580 with two 2.8 GHz Xeon™ processors (512 KB L2, 2 MB L3, hyperthreading disabled), 32 GB RAM, and four Intel® PRO/1000 gigabit network interfaces (NICs). Eight HP Integrity rx2600 machines act as clients. A switch interconnects all the machines, and the network is configured to support two clients on each server NIC's subnet. Each of these machines is also connected to a control machine on a distinct control network, which is used to control experiments and collect results.

Our ETA+AIO experiments dedicate one CPU to the PPE and the second (host) CPU runs one instance of the userver. For our experiments with the standard Linux stack (*baseline*), we use best-practice configurations to maximize baseline system performance. Specifically, the baseline configurations bind a userver process and the interrupts from the interfaces it listens on to the same CPU [8]. For our two-CPU server, each CPU runs a single userver process which listens on the IP addresses of two of the four network interfaces. We use `epoll` in the baseline, which we found to be more efficient than `poll` or `select`. We configure each NIC to use checksum offloading, and for baseline configurations we also enable the dynamic interrupt throttling capability of each NIC to minimize interrupt overhead. We do not use TCP segmentation offload (TSO) because it is not supported by ETA+AIO's 2.4.16-based networking stack. We would expect TSO to provide comparable benefits for ETA+AIO and the baseline.

7.2 Overall Network Processing Performance

To obtain a high-level characterization of the performance benefits of ETA+AIO, we evaluate the ETA+AIO prototype and baseline configurations in terms of two key performance attributes: *network processing efficiency*, which we define as the fraction of CPU cycles that are available for additional application processing, and *peak throughput*. For any fixed level of delivered throughput, configurations with lower

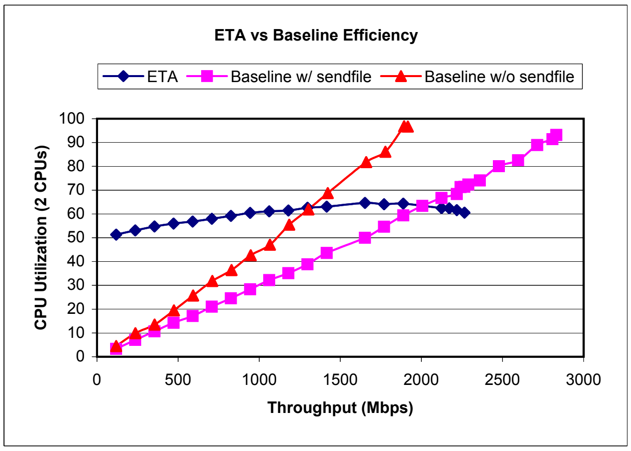


Figure 4: Processing Efficiency and Capacity

CPU utilization have higher network processing efficiency, since they leave a larger fraction of CPU cycles available for additional application processing.

Figure 4 plots the CPU utilization (aggregate of the two CPUs, measured with `vmstat`) required to achieve each level of throughput for ETA+AIO and two baseline configurations, using the userver and the SPECweb99-like workload. For each configuration, the rightmost data point corresponds to the peak throughput achieved with that configuration. This value represents the actual system capacity, free of artifacts like AIO error cases, which never occur in these experiments. We compare ETA+AIO against baseline configurations that use the Linux 2.4.18 kernel with a processor affinity patch, since the 2.4.18 kernel is the basis for the ETA+AIO prototype[‡]. In one baseline configuration the userver application uses the `write` system call to transmit data as is commonly the case when sending dynamic content instead of static content. The `write` system call incurs the overhead of a memory-to-memory copy from the user data buffer to kernel socket buffers. In the other baseline configuration, the userver application uses the "zero-copy" `sendfile` system call to transmit data without memory-to-memory copies.

Figure 4 shows that, as expected, network processing efficiency and peak throughput in the baseline substantially improve with the use of `sendfile`. The baseline without `sendfile` achieves a peak throughput of 1916 Mbps, while the baseline with `sendfile` achieves 2832 Mbps. These results reflect the performance capabilities of the kernel and network stack implementations since the peak throughput is limited by the saturated CPU rather than bottlenecks in the memory or I/O subsystems.

The results for ETA+AIO in Figure 4 show that its aggregate CPU utilization is always at least 50%, because in our server one of the two CPUs runs the continuously polling PPE. Thus, at low network throughputs when the PPE CPU is not well-utilized, the network processing efficiency of the ETA+AIO system is considerably worse than the baseline configurations. In contrast, at high network loads the PPE CPU is effectively utilized enabling ETA+AIO to have higher network processing efficiency than the baseline configurations. In addition, for future systems with a larger number of processor cores, a single PPE will comprise a lower percentage of total CPU cycles than 50%, likely narrowing the region of light load levels where ETA+AIO has worse network processing efficiency.

[‡]We compared a 2.6 kernel's networking to the 2.4.18 kernel and saw only a small performance increase.

Compared to the baseline without `sendfile`, ETA+AIO achieves higher network processing efficiency over a wide range of throughput (higher than 1300 Mbps) and also supports a higher peak throughput (2266 Mbps versus 1917 Mbps). These improvements can be largely attributed to the elimination of transmit-side copies in ETA+AIO. This result suggests that applications that are not able to use copy-avoidance primitives such as `sendfile` can see significant performance benefits with ETA+AIO.

ETA+AIO also achieves higher network processing efficiency than the baseline with `sendfile`, but over a narrower range of high throughput starting at 2000 Mbps. For example, at 2266 Mbps throughput, ETA+AIO has 9% more cycles available for application processing than the baseline with `sendfile` (62% versus 71% utilization at the same throughput). The efficiency improvement shows that additional architectural aspects of ETA+AIO besides transmit copy elimination benefit network processing. Despite these improvements, the ETA+AIO prototype achieves lower peak throughput than the baseline with `sendfile` (2266 Mbps versus 2832 Mbps). Although the ETA+AIO system has a substantial fraction of idle CPU cycles at its peak throughput, the PPE CPU is saturated, preventing additional network processing. However, these cycles can be utilized for other application processing, potentially allowing ETA+AIO to support higher throughputs for an application with a larger component of application computation (e.g., a webserver generating dynamic content).

To study this effect, we next evaluate performance on a workload that includes the encryption of a subset of the files. We extend the userver to encrypt a configurable percentage of the files that are at most 7500 bytes in size. In these experiments, to maximize efficiency the baseline system uses `sendfile` to transmit all data, including the dynamically encrypted files.

Figure 5 shows the peak throughput and CPU utilization for various levels of encryption. Since our ETA+AIO prototype statically allocates the application and PPE CPUs, it does not compare favorably to the baseline when the CPU demand is biased toward either networking or the userver application. However, the ETA+AIO prototype achieves higher throughput than the baseline for encryption percentages in the range between 25% and 55%. Furthermore, within this range the ETA+AIO prototype often has lower CPU utilization than the baseline.

7.3 Detailed Performance Analysis

Section 7.2 identified workload regions in which the ETA+AIO prototype has a modest advantage over the best baseline configuration in throughput and efficiency. We next present a detailed analysis to identify and quantify the reasons for the observed efficiency advantages.

Whereas our prototype assigns a single CPU for use by the networking stack (PPE), with the 2-CPU baseline system there may be an instance of TCP/IP executing on each processor. Due to cache conflicts and semaphore contention, TCP/IP tends to scale less than linearly on baseline systems

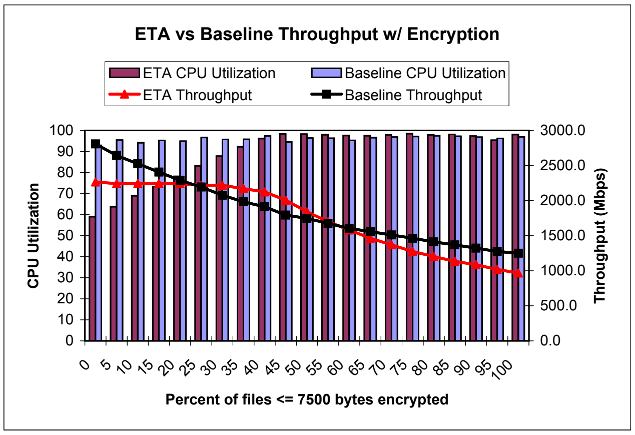


Figure 5: Peak Throughput and CPU Utilization with an Application Workload

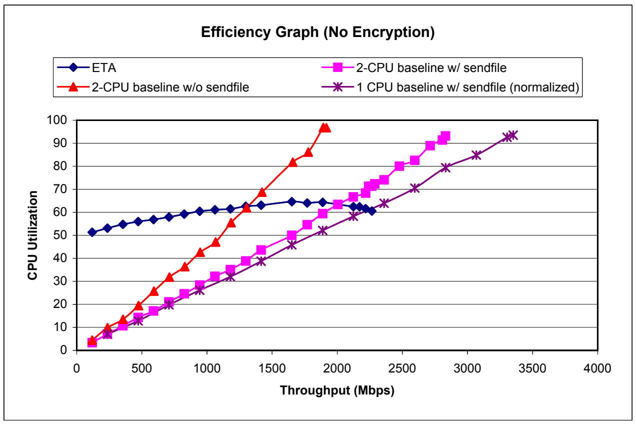


Figure 6: Processing Efficiency with Normalized 1-CPU Data

with multiple CPUs. Therefore, it is important to determine what portion of the ETA+AIO prototype's benefit is a result of running only a single instance of the networking stack. We conduct experiments to understand this effect, and so that we may factor it out when comparing the baseline to the ETA+AIO architecture.

To evaluate the benefits of performing packet processing on only one CPU, we measure a baseline configuration executing with only one CPU enabled. For this configuration, we use only two NICs rather than four and execute at half the HTTP request rate of the 2-CPU systems. We normalize the 1-CPU results to the ETA+AIO and 2-CPU results by doubling the 1-CPU system's observed throughput. This normalized 1-CPU baseline provides a best-case representation of an idealized 2-CPU baseline which would be free of the effects of multi-processor networking stack contention.

In Figure 6 we compare the normalized 1-CPU baseline (utilizing `sendfile`) against the real 2-CPU baselines and the ETA+AIO prototype. At the peak throughput for ETA+AIO, CPU utilization roughly matches the 1-CPU baseline. Hence it appears that much of the benefit of our prototype versus the 2-CPU baseline does come from avoiding multiprocessing overheads for packet processing.

Though for this workload the ETA+AIO prototype is no more efficient than the 2-CPU baseline after discounting the multiprocessing overhead, we wish to determine if the two architectures are exhibiting equivalent behavior. For example, features of the ETA+AIO architecture may be providing efficiency benefits that are offset elsewhere. To understand how CPU cycles are distributed in the ETA+AIO architecture versus the baseline architecture, we collect data from the hardware performance counters using oprofile [2]. We choose a target rate of 18800 requests per second, which results in a throughput of 2218 Mbps. This is slightly less than the peak ETA+AIO throughput because peak throughput is slightly degraded when oprofile sampling is enabled. We group the oprofile results into the following bins:

Userver Includes the userver code plus time spent in libraries called by the userver, including libc, the DUSI library, and the AIO mapping layer.

TCP/IP Contains functions we are able to attribute to TCP/IP processing. In order to ensure that the ETA+AIO and baseline TCP/IP bins include only similar functionality, the baseline TCP/IP bin excludes socket system calls.

Ethernet Contains the NIC device driver and general device processing that we directly attribute to the NICs (or is dominated by the NICs such that other devices are insignificant). For the baseline, this includes the cost of NIC interrupts; for ETA+AIO, it includes the cost of NIC polling.

Kernel other A catchall bin for kernel processing that does not fall into the other bins. This includes functions that cannot be attributed to the other bins (e.g., memory allocation, where we cannot reliably determine which calls are a result of TCP/IP, the Ethernet driver or elsewhere). Socket and file-related functions are included in this bin.

PPE other Consists of oprofile samples consumed by the PPE that are not directly attributable to TCP/IP or Ethernet processing. This bin includes the polling of the application queues.

Table 1 shows the measured distribution of CPU cycles for the workload without encryption. These results show that the ETA+AIO system has a nearly identical percentage of idle cycles compared to the idealized and normalized 1-CPU system (39.24% versus 40.08%), and a significantly greater percentage of idle cycles than the actual 2-CPU baseline system (29.30%). The ETA+AIO idle cycles are all on the host CPU since the PPE CPU is continuously polling. In addition to the polling of NICs attributed in the *Ethernet* bin, the PPE is also continuously polling the application's DbQ and the Administrative Queue (Figure 2). The cost of the application polling manifests itself in the *PPE other* bin. Our examination of the oprofile details indicates that approximately half of the *PPE other* cycles in Table 1 are consumed by unproductive polling.

The results for the TCP/IP bin in Table 1 show that the ETA+AIO prototype executes TCP/IP more efficiently than the 2-CPU baseline with `sendfile`. However, the ETA+AIO prototype and the normalized 1-CPU system have very similar TCP/IP performance. As discussed earlier, the ETA+AIO prototype's TCP/IP stack benefits from

Table 1: ETA+AIO vs Baseline CPU Utilization

Bin	*ETA+AIO*	*Baseline 2-CPUs*	*Baseline 1-CPU*
Idle	39.24%	29.30%	40.08%
Userver	6.89%	5.48%	5.32%
TCP/IP	25.02%	29.51%	25.54%
Ethernet	15.01%	19.19%	16.82%
Kernel other	8.31%	16.40%	12.09%
PPE other	5.43%		

executing on just a single processor. It is not surprising that the ETA+AIO *TCP/IP* bin is not noticeably more efficient than that of the 1-CPU system when there is no encryption workload. Since we have not changed the TCP/IP stack, the only expected improvement would be due to better cache behavior. On our measurements of encryption workloads we do observe small but consistent improvements in ETA+AIO TCP/IP efficiency over a 1-CPU system. For example, when we sampled a 15400 request per second workload encrypting 40% of files not greater than 7500 bytes, the ETA+AIO system spent 20.05% of its time in TCP/IP, while the 1-CPU system consumed 21.63% of the CPU cycles.

Table 1 also shows that the baseline spends more time in the kernel (16.40% for the 2-CPU system and 12.09% for the 1-CPU system) than the ETA+AIO system (8.31%). An analysis of the detailed oprofile data shows that most of the baseline's kernel overhead is attributed to sockets and `epoll`. In addition, this analysis shows that signals and synchronous DUSI events (e.g., creating and destroying DTIs) are a substantial part of the ETA+AIO architecture's kernel overhead. Recall that the application uses signals only when it must wait for a new event to arrive to empty event queues. This requires the userver to make several signal-related system calls. In addition, any DUSI synchronous calls invoke an `ioctl` system call to schedule the ETA Kernel Agent. When the PPE raises a signal or completes a synchronous event, the Kernel Agent is scheduled via an Interprocessor Interrupt (IPI). Though the AIO interface provides significant benefits over sockets and `epoll`, the gains are partially mitigated by the overhead of signals waiting for events. Our analysis of the oprofile data shows the combined cost of signal-specific kernel functions, the ETA kernel agent, IPIs and system calls to be 2.70% of the system's CPU cycles, or approximately a third of the ETA+AIO kernel overhead. In Figure 4 we note that as ETA+AIO begins to reach its peak throughput it becomes *more* efficient. This is because at high loads the userver is more likely to find an event on its queues, and thus less likely to incur the overhead of signals.

Table 1 shows that the ETA+AIO Ethernet component consumes less CPU cycles (15.01%) than both the 2-CPU and 1-CPU baseline Ethernet components (19.19% and 16.82%, respectively). This is despite the PPE's continuous polling of NICs, which consumes CPU cycles even when no packets are available. A more detailed analysis of the oprofile data reveals that the ETA+AIO architecture benefits from not having to incur the overhead of hardware interrupts, and from not having to go through the Linux SoftIRQ path. However, these operations are quite efficient on the baseline system, accounting for only slightly more than 2% of the overall system overhead. The low cost of interrupts on the baseline system is likely due to the effective use of NIC interrupt moderation timers.

Table 2: ETA+AIO vs Baseline Cache Misses (No Encryption). Equal normalized throughputs

Cache	*ETA+AIO*	*Baseline 2-CPUs*	*Baseline 1-CPU*
L3 Misses	1.43	1.34	1.00
L2 Misses	1.01	1.16	1.00

Table 3: ETA+AIO vs Baseline Cache Misses (With Encryption). Equal normalized throughputs

Cache	*ETA+AIO*	*Baseline 2-CPUs*	*Baseline 1-CPU*
L3 Misses	1.00	1.38	1.00
L2 Misses	0.71	1.13	1.00

Since ETA+AIO partitions the networking stack from the application with a goal of reducing resource contention, we use oprofile to evaluate the impact of partitioning on cache behavior. We note that the unified caches of the Xeon™ processor preclude measurement of separate cache behavior for instructions and data. Table 2 shows the relative number of all Level-3 (L3) and Level-2 (L2) cache misses for ETA+AIO, a 2-CPU baseline system and a normalized (half-rate) 1-CPU baseline system when the userver application has no additional computation to perform (i.e., executing without encryption enabled). The values are reported relative to the normalized 1-CPU baseline cache miss counts. For example, in Table 2 the 2-CPU baseline has 34% more L3 cache misses per CPU than the 1-CPU baseline. Table 3 presents the relative L3 and L2 cache misses for a workload where the userver does have additional computation to perform (i.e., encrypting 40% of the files not greater than 7500 bytes).

Table 2 shows that without additional application computation (no encryption) the ETA+AIO system has a higher L3 cache miss rate than both baseline configurations. This can be attributed to the cache-to-cache transfers between the host CPU and the PPE for the DTI interactions. However, Table 3 shows that with an encryption workload the rate of L3 cache misses between ETA+AIO and the 1-CPU baseline system is comparable, and much better than the 2-CPU baseline. An analysis of the oprofile data indicates that without encryption ETA+AIO incurs approximately 1273 L3 cache misses per megabit of data served, while the 1-CPU baseline incurs roughly 890 L3 misses per megabit. For the encryption workload, the ETA+AIO L3 cache miss rate increases slightly to about 1310 L3 misses per megabit, while the 1-CPU baseline system's L3 cache miss rate increases over 45% to approximately 1310 misses per megabit. The additional encryption workload increases the cache contention between the application and the network stack on baseline systems. When a file is encrypted, the userver application reads the original file and writes encrypted data to an encryption buffer, potentially causing evictions from the L3 cache. For a 1-CPU baseline system with an encryption workload, encrypting the data accounts for almost 17% of the L3 cache misses. For the ETA+AIO system an eviction in the application processor's L3 cache does not affect PPE packet processing, and thus processor partitioning reduces the impact of the additional application computation on the L3 cache miss rate. In contrast to the 1-CPU baseline, ETA+AIO encryption accounts for roughly only 10% of the total L3 cache misses.

Table 2 shows that ETA+AIO's L2 cache behavior is similar to the 1-CPU baseline when there is no encryption workload. However, as we add the encryption application workload (Table 3) the ETA+AIO system L2 cache behavior becomes substantially better than the baseline. We attribute this to the network stack and application partitioning, as we expect the ETA+AIO architecture to have better locality and a smaller per-CPU instruction working set compared to the baseline.

7.4 Summary of Results

Using a web server workload with a configurable computational component (encryption), our performance evaluation has characterized workload conditions where either the ETA+AIO prototype or the baseline provides superior performance or efficiency. Our detailed performance analysis provides insight into the reasons for this behavior. Our results show that the transmit-side copy avoidance in the prototype results in significantly better processing efficiency and higher peak throughput than a baseline system that uses the standard `write` system call to transmit replies to web clients. After factoring out the impact of copy avoidance by using a baseline system that uses the `sendfile` system call, the prototype retains a (smaller) efficiency advantage over the baseline. This efficiency advantage only partially offsets the fact that for packet processing, the baseline system uses two CPUs whereas the prototype uses only one (the PPE CPU). Hence, for workloads with lightweight application processing requirements, the prototype achieves a lower peak throughput than the `sendfile` baseline. For workloads in which application processing and packet processing better match the CPU allocation, the prototype can achieve marginally higher peak throughput than the baseline.

By comparing a one-CPU version of the `sendfile` baseline to the prototype, we show that most of the efficiency benefits of the prototype come from avoiding the overheads of multiprocessing for packet processing (since the prototype uses only a single PPE CPU). The detailed breakdown of CPU execution time shows that additional efficiency benefits come from the use of the AIO interface, which has lower overhead than the standard sockets interface. For memory-intensive application workloads, processor partitioning eliminates contention between the application and TCP processing for a shared processor cache, and hence reduces cache misses compared to baseline systems. Finally, the signal mechanism used by the application to wait for events when all the event queues are empty can introduce significant overhead and should be replaced by some lighterweight mechanism. However, this problem occurs only for intermediate traffic loads, since at low loads the CPU usage is negligible, and at high loads event queues are empty less often.

8. ARCHITECTURAL EXTENSIONS

Our analysis points to many interesting directions to extend the implementation and investigation of the ETA+AIO architecture. To support higher throughputs, the ETA prototype should be extended to scale to multiple PPE CPUs. An important question to answer is how PPE scaling affects the performance benefits of processor partitioning. For example, in our study the bottleneck was the CPU cycles available in the PPE (or the host CPU). By adding PPEs, the bottleneck resource can shift from CPU cycles to memory bandwidth, at which point the cache benefits of partitioning will likely become more important than we have observed in our investigation. With multiple PPEs, additional issues arise such as finding the best assignment of responsibility for the NICs in a system to the various PPEs. In order to keep up with the link speed of a single high-speed (e.g., 10 GigE) NIC, it may be necessary for multiple PPEs to share a single NIC, perhaps by spreading TCP connections for the NIC across the PPEs as proposed by Microsoft's receive-side scaling (RSS). For implementations that support multiple PPEs, the number of PPEs should be dynamically varied in proportion to the workload.

Scheduling issues are also interesting to investigate. AIO enables applications to present large numbers of concurrent operations to the PPE. In our prototype, some scheduling parameters in the PPE loop are chosen through trial-and-error, such as the number of entries to dequeue from a NIC in each loop iteration. Different choices may be optimal for different workloads, and a method for dynamically tuning the scheduling of PPE operations may yield higher throughput. In addition, application performance may depend on the order in which the application processes events.

The benefits of processor partitioning may be greater for emerging multi-core processor architectures than in our prototype, for example due to on-chip mechanisms for fast communication between processor cores. In addition, network interface logic may become integrated onto the same chip as the processor cores, avoiding the costs of traversing off-chip I/O links such as PCI. It is important to understand how this integration would affect the performance behavior of the ETA+AIO and baseline architectures.

9. CONCLUSIONS

The high cost of TCP/IP packet processing and the emergence of high-speed network links such as 10 gigabit Ethernet pose an important challenge for next-generation systems. In this paper we presented the design and evaluation of a prototype implementation of the ETA+AIO architecture which addresses this problem by dedicating processors to packet processing and by exposing asynchronous I/O (AIO) semantics to applications. We also presented a case study showing how the use of asynchronous network socket primitives affects the design of a real server application.

Compared to previous work, the evaluation presented in this paper provides a more accurate and comprehensive assessment of the potential benefits of approaches that dedicate CPU cores to networking, particularly in light of recent advances in OS and TCP stack implementations. Prior investigations of ETA and a similar architecture (TCP Servers) have the disadvantage that they only compared their approaches against baseline systems that did not use best-practice configurations and did not use OS features that have recently become available to improve performance. In contrast, our evaluation makes full use of current best-practice techniques including use of the `sendfile` system call, processor affinity for both processes and interrupts, and NIC hardware support for interrupt moderation and TCP checksumming. In our experience, use of these features for the baseline system significantly improves performance, as reported in recent literature (e.g., [19]). Therefore, the results of our study differ substantially from previous results, showing more nuanced benefits for processor partitioning than have prior investigations which have claimed improvements in peak throughput on the order of 30% [42, 44, 41].

Our results (summarized in Section 7.4) show that the ETA+AIO prototype provides marginally greater processing efficiency than standard systems by avoiding multiprocessing overheads for packet processing and by using the lower overhead AIO API instead of sockets. Also, processor partitioning reduces cache misses for memory-intensive applications. For workloads in which application processing and packet processing approximately match the CPU allocations, the ETA+AIO prototype effectively utilizes both the host and PPE CPUs, leading to higher peak throughput using fewer CPU cycles.

In addition to presenting the performance evaluation, we also described in detail the issues involved in using asynchronous socket I/O to implement an event-driven web server application. We described the construction of the finite-state machine which handles each connection. We determined that only a subset of the event types is needed to drive the state machine operation, while other event types can be ignored. We also discussed the challenges of managing memory registration and de-registration, and the scheduling opportunities provided by the ability to use multiple event queues. The userver implementation using the AIO API is open source software.

Although our prototype demonstrated only modest performance advantages over the well-configured baseline system, our investigation motivates a number of additional investigations as outlined in Section 8 which may lead to greater performance improvements in the future. These investigations include scaling to multiple PPEs, changing PPE and application scheduling policies, and exploiting features of multi-core processors like fast inter-processor communication which may improve the benefits of partitioning.

Acknowledgements

The authors gratefully acknowledge the contributions of Paul Stillwell, Jr., John Wiegert, Phil Cayton, Jeff Gray, and Greg Regnier of Intel® Corporation on the design and development of the ETA software prototype. In addition, Tim Brecht gratefully acknowledges the financial support of Hewlett-Packard, the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada. Furthermore, the authors would like to acknowledge the efforts of the reviewers and our shepherd Alan Cox for their insightful comments that have improved the clarity of the paper.

10. REFERENCES

[1] Apache. URL `www.apache.org`.

[2] OProfile. URL `oprofile.sourceforge.net/news/`.

[3] RDMA Consortium. URL `www.rdmaconsortium.org`.

[4] Sockets API Extensions. URL `www.opengroup.org`.

[5] Zeus Technology. URL `www.zeus.co.uk`.

[6] Design notes on asynchronous I/O (aio) for Linux, 2002. URL `lse.sourceforge.net/io/aionotes.txt`.

[7] The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2003 Edition.

[8] V. Anand and B. Hartner. TCP/IP network stack performance in Linux kernel 2.4 and 2.5. In *Proceedings of the Linux Symposium*, pages 8–30. Ottawa, Ontario, Canada, July 2003.

[9] B. S. Ang. An evaluation of an attempt at offloading TCP/IP processing onto an i960rn-based NIC. Technical Report HPL-2001-8, HP Labs, Palo Alto, CA, Jan 2001.

[10] G. Banga, J. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA, June 1999.

[11] A. V. Bhatt. Creating a PCI Express interconnect. URL `www.pcisig.com/specifications/pciexpress/technical_library/pciexpress_whitepaper.pdf`.

[12] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Performance analysis of system overheads in TCP/IP workloads. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. St. Louis, September 2005.

[13] T. Brecht and M. Ostrowski. Exploring the performance of select-based internet servers. Technical Report HPL-2001-314, HP Labs, November 2001.

[14] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference*. Boston, June 2004.

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

[16] Z. Ditta, G. Parulkar, and J. Cox Jr. The APIC approach to high performance network interface design: Protected and other techniques. In *Proceedings of IEEE INFOCOM '97*, volume 2, pages 7–11, April 1997.

[17] 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):66–76, March-April 1998.

[18] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the 2004 USENIX Annual Technical Conference*. Boston, June 2004.

[19] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *IEEE International Conference on Networks*, November 2004.

[20] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance re-visited. In *IEEE International Symposium on Performance of Systems and Software*, March 2003.

[21] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server network scalability and TCP offload. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 209–222. Anaheim, April 2005.

[22] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of 1999 USENIX Technical Conference (Freenix Track)*, pages 109–120, June 1999.

[23] J. M. Hart. *Win32 System Programming.* Addison Wesley, 2nd edition, 2001.

[24] HP Labs. The userver home page, 2005. URL `www.hpl.hp.com/research/linux/userver`.

[25] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32nd International Conference on Computer Architecture (ISCA'05).* Madison, WI, June 2005.

[26] InfiniBand^{SM} Trade Association. *InfiniBand^{TM} Architecture Specification Volume 1, Release 1.0.* October 2000. URL `www.infinibandta.org`.

[27] Intel® Corporation. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual, Revision 2.5.* July 2005.

[28] V. Jacobson and B. Felderman. A modest proposal to help speed up and scale up the linux networking stack. In *linux.conf.au*, January 2006.

[29] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *SIGCOMM*, pages 259–268, 1993.

[30] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transations on Networking*, 4(6):817–828, 1996.

[31] Y. Khalidi and M. Thadani. An efficient zero-copy I/O framework for UNIX. Technical report, SMLI TR95-39, Sun Microsystems Lab, May 1995.

[32] D. Libenzi. Improving (network) I/O performance. URL `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[33] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX).* USENIX, May 2003.

[34] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[35] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. Madison, WI, June 1998.

[36] S. Muir and J. Smith. AsyMOS – an asymmetric multiprocessor operating system. In *IEEE Conf on Open Architectures and Network Programming (OPENARCH)*, April 1998.

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

[38] S. Nagar, P. Larson, H. Linder, and D. Stevens. epoll scalability web page. URL `http://lse.sourceforge.net/epoll/index.html`.

[39] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[40] M. Rangarajan, K. Banerjee, J. Yeo, and L. Iftode. MemNet: Efficient offloading of TCP/IP processing using memory-mapped communication. Technical Report DCS-TR-485, Rutgers University Technical Report, 2002.

[41] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. TCP Servers: Offloading TCP processing in Internet servers. Technical Report DCS-TR-481, Rutgers University, Mar 2002.

[42] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. ETA: Experience with an Intel® Xeon™ processor as a packet processing engine. In *Hot Interconnects*, August 2003.

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

[44] V. A. Saletore, P. M. Stillwell, J. A. Wiegert, P. Cayton, J. Gray, and G. J. Regnier. Efficient direct user level sockets for an Intel® Xeon™ processor based TCP on-load engine. In *The Workshop on Communication Architecture for Clusters.* Denver, CO, April 2005.

[45] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *5th Annual Linux Showcase and Conference*, pages 165–172, November 2001.

[46] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: when does hardware support help? In *2nd USENIX Conference on File and Storage Technologies (FAST)*, Mar 2003.

[47] P. Shivam and J. S. Chase. On the elusive benefits of protocol offload. In *ACM SigComm Workshop on Network-IO Convergence (NICELI).* Germany, August 2003.

[48] Standard Performance Evaluation Corporation. *SPECweb99 Benchmark*, 1999. URL `www.spec.org/osg/web99`.

[49] W. Stevens. *Unix Network Programming, Volume 1.* Addison Wesley, third edition, 2003.

[50] Y. Turner, T. Brecht, G. Regnier, V. Saletore, G. J. Janakiraman, and B. Lynn. Scalable networking for next-generation computing platforms. In *Third Annual Workshop on System Area Networks (SAN-3).* Madrid, Spain, February 2004.

[51] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. In *18th Symp. on Operating System Principles (SOSP-18)*, Oct 2001.

[52] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.