Appears in the Proceedings of the Third Annual Workshop on System Area Networks (SAN−3),
Madrid, Spain, February, 2004.

Scalable Networking for Next-Generation Computing Platforms

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

*Hewlett-Packard Laboratories Palo Alto, CA
§Intel Corporation Hillsboro, OR
‡University of Waterloo Waterloo, Canada

***Abstract*— We propose a technology strategy for enabling applications to scale to next-generation levels of I/O scalability and communication performance on industry standard platforms. The strategy combines efficient packet processing and scalable I/O concurrency, potentially enabling Ethernet and TCP to approach the latency and throughput performance offered by today's System Area Networks. We target the performance of communication-centric applications, initially using a web server as the application for concept validation.**

Our approach integrates two components to provide a complete networking stack for user-level applications. The first component is the ETA architecture developed at Intel Labs, where one or more server processor cores are dedicated for network packet processing. This architecture reduces processing overhead by avoiding costly interrupts and context switches, and by exposing VIA-style user-level communication primitives which reduce data copies and bypass the operating system for most I/O operations. The second component is an asynchronous I/O (AIO) programming model, which allows a single thread to issue I/O operations without being blocked and to receive asynchronous events that signify the completion of previously issued I/O operations. The AIO programming model allows applications to achieve scalable concurrency without the overhead of software threads.

This paper will describe the components of our proposed networking stack, a web server application designed to take advantage of this networking stack, and our plans to evaluate the performance benefits of this approach.

I. Introduction

Future computing platforms will require fast, low overhead packet and protocol processing to accomodate emerging high speed I/O interfaces such as 10 gigabit Ethernet. At high packet and data rates, current systems suffer high overhead from interrupts, context switches, and data copies. These overheads hurt application performance by consuming processor and memory bus cycles, polluting caches, and disrupting processor pipelines.

Future platforms will require increased I/O concurrency, in addition to efficient packet processing, to take full advantage of new high speed links. As improvements in I/O bandwidth outstrip improvements in latency, an increased number of operations must be in flight concurrently to effectively utilize link bandwidth. However, conventional software multithreading approaches to increase concurrency scale poorly, with intolerable overhead for scheduling and coordination [1].

In this position paper, we propose a technology strategy for enabling applications to scale to next-generation levels of I/O performance on industry standard platforms. The strategy combines efficient packet processing and scalable I/O concurrency, potentially enabling Ethernet and TCP to approach the latency and throughput performance offered by today's System Area Networks (SANs). We target the performance of network-centric applications, using a web server as the initial application for concept validation.

The proposed approach integrates two components to provide a complete networking stack for user-level applications:

- **Embedded Transport Acceleration (ETA)** [2]. The ETA project at Intel Labs has developed a prototype architecture in which all network packet processing is segregated to one or more dedicated processors or hardware threads that are fully integrated in the system's cache coherence domain. These dedicated Packet Processing Engines (PPEs) interact with network interfaces and with applications via load/store instructions to cache-coherent memory, avoiding costly interrupts and context switches. The architecture exposes VIA-style [3] user-level communication to applications, reducing data copies and bypassing the operating system for most I/O operations [4][5].
- **Asynchronous I/O (AIO) API**. To achieve scal-

able concurrency without the overhead of software threads, we adopt APIs that expose high-level file and socket asynchronous I/O (AIO) semantics to applications [6][7][8][9]. The APIs support asynchronous versions of traditional socket and file operations. Applications issue operations without being blocked and without first verifying file and socket descriptor status. Applications receive asynchronous *events* that signify the completion of previously issued I/O operations. This model allows multiple I/O operations to be in-flight concurrently. We initially focus on ETA support for socket AIO and rely on existing OS support for file AIO. We envision that ETA could support file AIO in the future using network file system protocols such as Direct Access File System (DAFS) [10].

The ETA architecture leverages microarchitecture trends toward multiple processor cores on a single die and multiple hardware threads per core [11][12]. Devoting a small number of cores to packet processing can be worthwhile in exchange for large improvements in application performance on the remaining hardware. Using standard processors ensures that the architecture will automatically track dominant semiconductor cost-performance trends (Moore's Law). Specialized processors and custom logic, as used in most TCP Offload Engines (TOEs), are in danger of falling behind the curve [13].

The ETA architecture, with fully programmable PPEs, can readily support evolving data center I/O functionality. PPEs based on standard processors can leverage their mature, widely used software development environments to facilitate rapid development and implementation upgrades. Wide deployment of commodity high speed links and the universal IP protocol will allow data centers to consolidate onto a cost-effective unified I/O fabric that supports networking, storage, and inter-process communication. PPEs can support a unified fabric by efficiently processing TCP/IP, upper layer protocols such as iSCSI for storage, and network data transformations such as encryption or compression.

By layering high-level asynchronous I/O APIs above the ETA architecture, our approach enables the development of highly concurrent, low overhead, event-based network-centric applications. There are two basic approaches to structuring applications to scale to highly concurrent operation: threads [1] and events [14]. Although good scaling can be achieved with special-purpose thread packages [1], no general purpose software threading package is currently available that can scale efficiently to huge numbers of threads. In the alternative event-based approach, the operating system or I/O devices deliver *events* to applications to indicate changes in system status (e.g., completion of an operation, or readiness of a file descriptor to perform a write operation without blocking). Applications are structured as state machines in which a state transition occurs when the application processes an event [14].

Using a high-level asynchronous I/O API, event-based applications can issue socket and file operations by placing operation descriptors into *work queues*. When an operation completes, a completion descriptor is placed by the PPE or OS kernel into an *event queue* accessible to the application. An application processes events to determine which operations to issue next. By exposing work and event queues to applications, the programming model naturally enables applications to process events and issue operations using application-specific scheduling policies to improve resource management and performance [15].

The remainder of this paper is organized as follows. Section II briefly presents background and related work. Section III gives an overview of the ETA architecture. Section IV discusses asynchronous I/O API alternatives and presents our API implementation. Section V describes the userver [16], our highly instrumented and flexible experimental web server we will use to validate our approach. Section VI discusses the research questions we plan to investigate as we build and evaluate a prototype implementation.

II. Background and Related Work

Most efforts to increase server network performance have centered on specialized TOE (TCP/IP Offload Engine) devices [17][18]. TOE devices generally offload varying amounts of the TCP/IP protocol stack on a device that attaches to the I/O subsystem of a server. TOE devices generally utilize separate, specialized processing and memory resources. The ETA prototype described in this paper differs from these devices in that it utilizes processing and memory resources of the server itself, making the packet processing engine a first class citizen of the core CPU and memory complex.

Other related research efforts include the QPIP [19] work at Berkeley that showed the effectiveness of interfacing IP protocols implemented on an intelligent network adapter using the Queue Pair model of the Infiniband™ Architecture [4]. The TCP Servers project [20] at Rutgers University showed a framework where the network processing could be partitioned onto

a dedicated node, processor or an intelligent adapter and interface to the host applications through lightweight communication mechanisms. TCP Servers is similar to ETA in terms of a partitioned architecture, but it differs in terms of interface functionality between the host and the Packet Processing Engine.

Several asynchronous I/O API implementations exist today, notably Microsoft's Windows asynchronous I/O with completion ports [9] and POSIX AIO [6], both of which support socket I/O and file I/O, and Linux AIO [7][21] which supports file I/O. These APIs are directed towards OS-based I/O stack (socket or files) implementations and, hence, do not have the primitives necessary for bypassing the OS. Recently, the Open-Group has begun to define asynchronous Sockets API Extensions [8] for Unix designed for implementations where I/O operations bypass the OS (e.g., with intelligent InfiniBand host adapters).

III. Overview of the ETA Architecture

The ETA architecture [2] partitions the server into the host and the packet processing engine or PPE. The host partition is where the operating system and applications execute. The PPE includes the processing and memory resources used for network-centric tasks, including TCP/IP protocol processing. The interface between the host and the PPE is implemented as a set of asynchronous queues in cache-coherent, shared host memory. The queuing structures are used for control, synchronization and for receiving and transmitting data on the network. Figure 1 shows the ETA architecture. Our ETA development vehicle is a multiprocessor server, where one processor is a dedicated PPE, and the remaining processors serve as the host.

ETA also defines the interface between the host and the PPE. This interface is implemented through a set of queuing structures called the Direct Transport Interface or DTI (see Figure 2). The ETA interface is based on the principles of the Virtual Interface Architecture [3] and Infiniband [4], but have been optimized for TCP streams and connection semantics. In particular, the DTI structures directly support the socket connection interfaces, and support the buffering semantics of TCP streams. Each DTI consists of a send queue, a receive queue, an event queue and doorbells. The send queue is used to post data transmit operations, in the form of a descriptor, to the PPE. The receive queue is used to post, or pre-post application buffers for incoming data. The doorbells are used to inform the PPE that new work has been queued for the PPE to process. The event queue

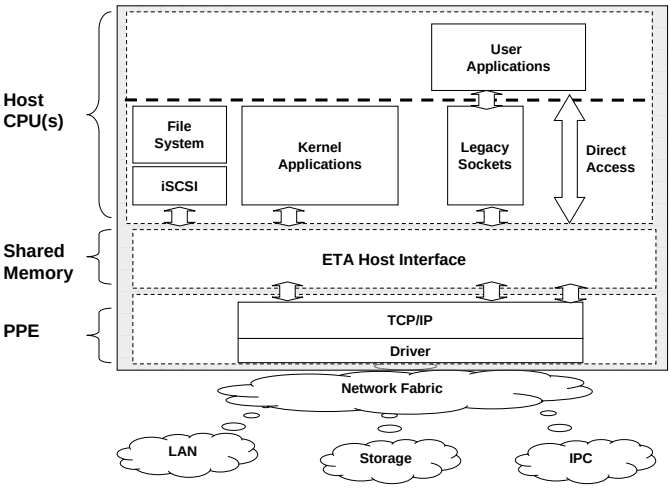

Fig. 1. ETA partitioned architecture

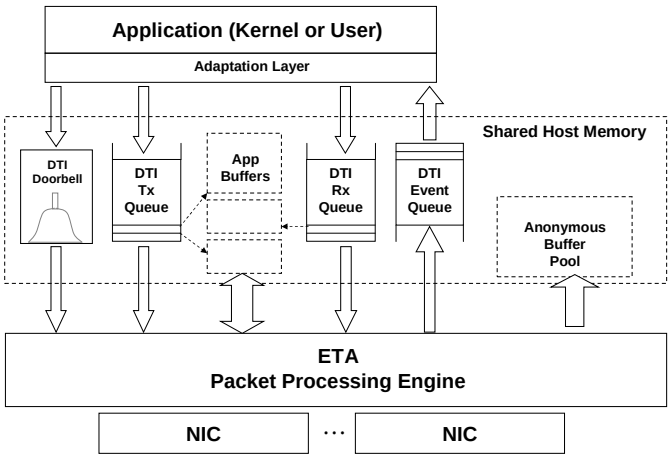

Fig. 2. DTI queuing structure

is the mechanism for synchronization, mainly informing the application when operations have completed. ETA also provides anonymous buffer pools in the shared memory in order to support buffering semantics of TCP streams by handling non-pre-posted or out-of-order receive packets. Finally, DTI event queues and doorbells can be shared across multiple DTI instances.

In order to achieve maximum performance, the ETA interface supports a direct access socket interface to user level applications. The direct access interface allows applications to initiate network operations that bypass the operating system and its associated overheads. Our implementation of this interface is a thin software layer called the Direct User Socket Interface or DUSI. DUSI supports asynchronous connection operations (connect, listen, accept) as well as send and receive data transfer operations.

IV. API FOR ASYNCHRONOUS I/O

In addition to using ETA for low overhead messaging, the second component of our approach is to provide APIs for asynchronous socket and file I/O. Asynchronous I/O (AIO) splits each I/O operation into two distinct phases: posting an operation request, and retrieving completion event information. An application thread can execute a non-blocking call that posts a request for an I/O operation to be performed. When the I/O operation eventually completes, a completion event is delivered asynchronously to the application, typically via an event queue, to indicate the operation's result. Split-phase I/O processing enables an application thread to perform other processing in between posting an operation and receiving its completion event. The other processing could include posting operations and processing completions for different I/O operations, thus increasing the level of I/O concurrency compared to synchronous, blocking I/O.

Unlike synchronous I/O, AIO does not block the calling application thread. We also distinguish AIO from *non-blocking* I/O, in which the application ensures that I/O operations will not block by first checking file or socket status. For example, in a Unix system, a socket or file can be set to non-blocking mode. Applications can call `select()` or `poll()` to detect which read or write calls can be performed without blocking the caller. The calls that will not block can return with only partial results, for example because of reaching socket buffer limitations, in which case the operations need to be submitted multiple times. Calls to read or write that cannot be performed without blocking do not succeed, and return EWOULDBLOCK. In contrast to non-blocking I/O, AIO operations never block, and therefore applications do not need to perform checks in advance or deal with partial completion.

A. Our API Implementation Approach

The asynchronous I/O API layer above the ETA architecture could be based on any of the existing AIO APIs described in section II. In the case of Microsoft Windows' completion ports, POSIX AIO, and Linux AIO, extensions are needed to support OS bypass in ETA. These APIs would need to support calls for applications to register memory regions that are to be the sources and destinations of packet data. Event delivery mechanisms must also be made lighter weight by not relying on the operating system (e.g., POSIX AIO relies on signal queues for event delivery). In contrast, the Open Group Sockets API Extensions, which is in development, supports the necessary calls and is promising. Since DUSI is similar to the Sockets API Extensions and we wanted to avoid making ad-hoc extensions to an arbitrarily chosen API, we have chosen a path which leaves us open to the use of any of the existing AIO APIs on an ETA platform. To this end, our web server application is coded to a generic interface that can be mapped to various AIO APIs. This AIO mapping is intended to meet the requirements of the web server application while avoiding any features that are peculiar to specific AIO APIs. Currently, we are working to complete the mapping to DUSI which provides the set of essential socket operations and completion event delivery operations.

The basic operations provided by DUSI are shown in Table I. With DUSI, each socket is associated with one DTI structure. This one-to-one binding simplifies socket scheduling and resource management. DUSI provides the `DUSI_Socket()` call to create a socket. Separately, the `DUSI_Create_EvQ()` call is used to create event queues. Socket creation entails creating the socket's DTI structure, binding the DTI TX work queue to some event queue, and binding the RX work queue to either the same or some different event queue. The binding determines where completion events are delivered for operations that are posted to the work queues. In section V we discuss how we plan to explore using the event queue binding flexibility of DUSI to improve application performance.

AIO APIs differ in the granularity of event queue binding. Whereas with DUSI, each TX or RX work queue is bound to an event queue, with Windows, binding to an event queue (completion port) is at the granularity of a socket. With Linux AIO (and POSIX AIO, where event queues are implemented as POSIX realtime signal queues), event queue binding is at the granularity of each I/O operation as it is posted. With the Open Group Sockets API Extensions, the binding is based on the type of I/O operation (read, write, accept, etc.) for a socket. In addition, the Sockets API Extensions allows the binding to be changed dynamically, potentially before posting each operation. Dynamic binding gives this API the same effective granularity of event queue binding as with Linux AIO. Per-operation event queue binding can be useful in multi-threaded applications in which the operation of each thread is specialized, and completion events for operations posted by one thread should be processed by another thread. For example, the threads might form a processing pipeline in which each thread is assigned to process from a unique event queue [14]. When a thread posts an operation, it can specify that the completion event will be delivered to the event queue

TABLE I
BASIC DUSI OPERATIONS

Setup Operations:	
Dusi_Open_Ual()	Open an instance of an ETA User Adaptation Layer for user application for a specified Doorbell Queue size and Anonymous Buffer Pool size.
Dusi_Create_EvQ()	Create a DTI event queue of specified size in the attributes and return an opaque handle to the queue. The call can register a signal for the event queue specified in its attributes.
Socket Operations:	
Dusi_Socket()	Create a DTI socket per connection for a specified sized Rx and Tx work queues. Associate the Rx and Tx work queues with separate or common event queue.
Dusi_Connect()	Establishes a connection on the specified DTI for a specified destination address and port.
Dusi_Listen()	Listen for connections on a specified DTI.
Dusi_Bind()	Binds a DTI to a specific IP address, address length, TCP port, and address family.
Dusi_Accept()	Post an asynchronous accept operation for socket connection on the specified listening parent DTI socket and associate the new connection on the specified child DTI socket.
Dusi_Recv()	Post asynchronous receive operation on the specified DTI socket.
Dusi_Send()	Post asynchronous transmit operation on the specified DTI socket.
Dusi_Shutdown()	Shutdown or close a specified DTI socket connection.
Dusi_Set_Socket_Ops()	Set operational parameters for the specified DTI socket.
Dusi_Get_Socket_Ops()	Get operational parameters for a specified DTI socket.
Completion Event Operations:	
Dusi_Wait_All()	Wait on the specified DTI event queue and return an event Vector containing all the events completed so far and return the number of events completed. Block waiting for an event until Timeout period if the event queue is empty.
Dusi_EvQ_Num_Events()	Poll the specified DTI event queue and return the number of events on the event queue.
Dusi_Done()	Poll for an event on the specified DTI event queue.
Memory Registration:	
Dusi_Register_Memory()	Register a region of memory with ETA PPE.
Dusi_Deregister_Memory()	Deregister previously registered memory with the ETA PPE.

for the next thread in the pipeline. Although DUSI does not currently support per-operation event queue binding, the work queue-based event queue binding that it provides is adequate for the requirements of the web server application described in section V.

DUSI requires applications to perform *memory registration* on application memory regions prior to using them for asynchronous socket I/O operations. Memory registration invokes the operating system to pin a region in physical memory and provide the address to ETA to enable zero-copy transfers. The cost of OS invocations to register and de-register a memory region is amortized over the intervening operations that use the region. A second function of memory registration can be to issue access keys (called protection tags) to a memory region. If a region has a protection tag, ETA ensures it is accessed only by operations that present a matching protection tag.

Once a socket is created and memory regions have been been registered, the application can post socket AIO operations to DTI work queues for processing by an ETA PPE. Since this process bypasses the OS, it is efficient to

post an operation. In contrast, OS kernel-based AIO APIs may suffer high system call overhead unless multiple operations are batched into a single call. Linux AIO, for example, provides a batching mechanism [7]. Batching complicates the application design and can delay I/O operations if a work queue empties while a batch is being prepared for submission.

When operations complete, completion events are delivered by a PPE to the appropriate DUSI event queues as directed in socket creation. The application can retrieve the events by calling `DUSI_Wait_All()`. A callback mechanism is supported in which callback functions can be invoked for each event in the queue. In addition, applications can pass in a user-defined tag to an I/O operation, and the tag is returned to the application via the corresponding completion event.

A single application thread may need to process events from multiple event queues to maintain single-threaded operation while keeping separate the events of different types. For example, processing multiple event queues may be necessary if the application thread uses multiple AIO APIs (e.g., one queue for socket AIO, and another queue for file AIO). In an I/O intensive application, occasionally it is possible that all the queues of interest are empty. In this case, the application thread has no work to do and should block until an event arrives to any of the event queues. To support blocking on multiple queues, DUSI allows a user-specified signal to be registered with an event queue. When enabled, the signal is generated on transition of the event queue from empty to non-empty. When all queues are empty, the application can enable this signal generation and block on a call to `sigsuspend()`.

B. Unifying AIO for Files and Sockets

Although our initial focus is on providing socket AIO using DUSI and ETA, we also wish to ensure that applications can use file AIO together with socket AIO. File I/O is traditionally implemented in the OS kernel. In addition, we envision that the kernel file system could be layered on a block-level iSCSI layer implemented by ETA for network-based storage. We further envision that ETA could be used to bypass the kernel for network-based file I/O using protocols such as Direct Access File System (DAFS) [10].

Unless asynchronous file I/O operations can bypass the kernel, applications may need to wait on file I/O completion events from the kernel and on socket completion events from an ETA PPE. The signal mechanism described in section IV-A can facilitate waiting for multiple types of event queues. In addition, it may be useful to have unified event queues that can receive completion events for both files and sockets. This would require solving the problem of efficiently coordinating concurrent access to an event queue by both the PPEs (for sockets) and the OS kernel (for files). Another important issue is how ETA can support the popular Unix `sendfile()` call which allows a file to be transmitted to the network directly from the kernel-managed file buffer cache in the system memory without intermediate copies.

V. Examining a Web Server Application

To demonstrate the benefits of our approach we plan to experimentally evaluate the performance gains that can be obtained using a well known and widely used network-centric application, namely a web server. A web server is an excellent example of an application that can place extremely high demands on system resources, especially operating system resources. We believe that our approach can enable demanding applications such as web servers to multiplex among large numbers of active connections with significantly reduced costs which will significantly improve server performance.

Current approaches to implementing high-performance network-centric applications require special techniques for dealing with high levels of concurrency. This point is illustrated by first considering the logical steps taken by a web server to handle a single client request. Almost all network-centric applications follow similar steps. To simplify our illustration, our example assumes a persistent connection initiated by the client.

1) Wait for and accept an incoming network connection.
2) Read the incoming request from the network. If the client has no more requests, it closes its end of the connection and the server `read()` returns EOF, so close the connection.
3) Parse the request.
4) For static requests, check the cache and possibly open and read or mmap the file.
5) For dynamic requests, compute the result.
6) Send the reply to the requesting client.
7) Goto step 2

Figure 3 shows the socket calls that are required in the case where the request is for a file that is found in the cache.

Several of the steps listed above can block because they require interaction with a remote host, the network,

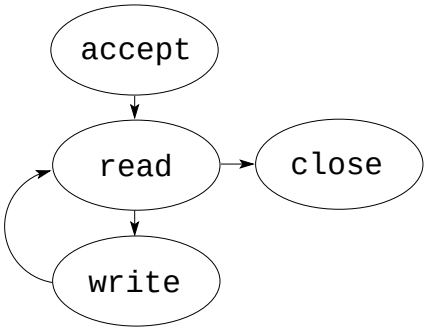

Fig. 3. Finite State Machine for Connections

a database or some other subsystem, and potentially a disk. Consequently, for high performance the server must handle several thousands or tens of thousands of simultaneous connections [22], quickly multiplex connections that are ready to be serviced, and dispatch network I/O events at high rates.

One approach to satisfy this requirement is to use a single process event driven (SPED) architecture (terminology from [23]) for the web server. A SPED architecture uses a single process and only issues calls that will not block [15][24]. This allows the process to service multiple connections concurrently without being blocked on an I/O operation. With a non-blocking I/O programming model, this requires using an event notification mechanism such as `select()`, `poll()`, or Linux's `epoll()` to determine when a system call can be made without blocking. In multiprocessor environments multiple copies of a single process event driven web server can be used to obtain excellent performance relative to alternative architectures [25].

An alternative approach multiplexes simultaneous connections by using processes or threads which naturally block in the operating system when a system call blocks. The multiplexing of connections occurs when the the operating system context switches to a thread that is ready to execute because an event that the thread was blocked on has now arrived. This is known as an MP (multi-process) or MT (multi-thread) model [26][23]. The Flash server implements an asymmetric multi-process event driven architecture (AMPED) [23]. This is a hybrid architecture combining event-driven socket I/O with helper processes dedicated to perform (blocking) disk accesses on behalf of the main event driven process. The capability to perform true asynchronous I/O favors the SPED model over alternatives based on threads.

A. The userver

To evaluate our architecture we will use an open source micro web server called the userver [15], [16], which is implemented using the SPED model. The userver currently supports a non-blocking socket I/O model and multiple event notification mechanisms (including Linux's `epoll()`), and it can run using multiple copies in multiprocessor environments. We will modify the userver to utilize asynchronous socket I/O. The userver has extensive tracing and statistics facilities which enables analysis of server behavior and the impact of various I/O and event notification interfaces on performance.

We plan to use the userver to compare the web server performance with and without the ETA engine. We will compare performance when using a dedicated processor for the ETA packet processing engine and the remaining processors for the execution of web server processes with the performance of a standard Linux implementation with web servers running on all available processors.

When using a standard Linux kernel and TCP/IP stack the separate userver processes will use the file system buffer cache to share cached files across all processes. Additionally, the Linux version of `sendfile()` is a zero-copy implementation which permits high-performance and a good basis for comparison. When utilizing the ETA packet processing engine the userver processes will share a file system buffer cache, although in this case this will be accomplished using `mmap()` to map files into the application's address space. Data will be transmitted using the DUSI API and the ETA engine without copying and without operating system involvement.

B. Web Server Event Scheduling

As noted in recent research an important issue in the design of web servers is the scheduling of events. For example, the balance between the rate at which new connections are accepted by the server and the rate at which forward progress is made on existing connections impacts server throughput [15][27]. If too few asynchronous `accept()` calls are initiated the server will not be accepting connections at a high enough rate. The server's listen queue will fill, and connection requests will be dropped. Conversely, if too many `accept()` calls are asynchronously initiated too many connections may be accepted and the server's performance will suffer because it's spending too much time accepting new connections and not enough time working on existing connections. Other researchers have noticed that the proper scheduling of asynchronous I/O reads and writes can significantly impact the performance of database queries [28].

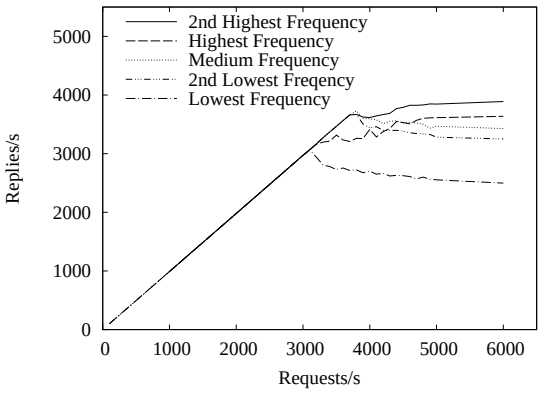

Fig. 4. Server throughput: impact of accept frequency

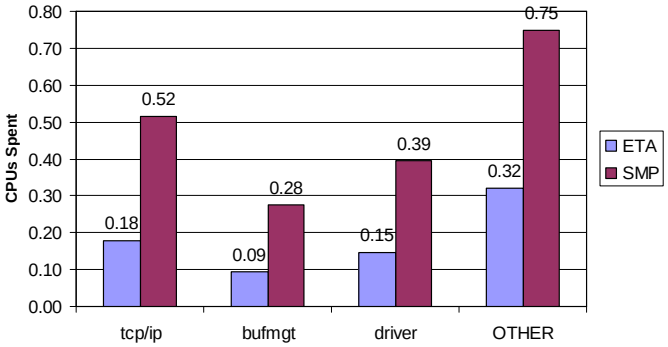

Fig. 5. ETA efficiency

To facilitate event scheduling in the userver, we will exploit DUSI's ability to deliver asynchronous I/O completion events of different types to separate completion queues. We plan to use DUSI to create and utilize multiple event queues: one to indicate when initiated `accept()` calls have completed, and additional queues for completed `read()`, `write()`, and `close()` calls. By combining the separation of queues by event type with a method for obtaining the number of events available in each queue the server can make informed decisions about what operations should be initiated next to obtain the balance required for peak performance.

Figure 4 demonstrates how this issue impacts the performance of a traditional SPED model server (i.e., the userver using a non-blocking model interacting with a standard Linux kernel). The server executes on a 500 MHz PIII-based HP NetServer LPr system running Linux. The client load is generated using `httperf` [29] and ten B180 PA-RISC machines running HP-UX 11.0 and consists of static requests conforming to the file size distribution used in SPECweb99 [30].

The graph in Figure 4 plots the throughput observed at the clients versus the targeted request rate. The different lines in the graph indicate different frequencies at which the server attempts to accept new incoming connection requests. These results confirm that server throughput suffers if the server accepts new connections too frequently or too infrequently. We believe that this issue will be important in our asynchronous I/O environment and will be a topic of our investigation.

VI. Evaluation Plan

We plan to evaluate our approach through a combination of measurement and analysis, as well as comparison to other design alternatives. We are building a functional prototype to use for this purpose. The prototype will build upon the existing ETA proof-of-concept with the addition of the Direct Socket User Interface (DUSI) to bypass the operating system in the performance critical path. In addition, we will provide an AIO mapping layer between the userver and the ETA DUSI interface.

The existing ETA proof-of-concept implementation uses a DUSI equivalent interface at kernel level. It is expected that DUSI will provide comparable performance to the kernel version of the interface due to the fact they both utilize the base DTI queuing and synchronization structures. Performance analysis on the existing kernel level interface shows that significant efficiency and performance gains can be obtained when comparing a dual processor server running ETA versus running in Symmetric Multi-Processing (SMP) mode. Figure 5 shows measured results of the kernel level prototype performing 1KB transmit operations to multiple client computers. It shows the relative amount of processing power consumed for various elements of the networking stack, including the TCP/IP code, the buffer management code, the driver code, and the remaining portions of the total stack. It clearly shows that each element of the ETA prototype stack is more efficient than its SMP counterpart.

The initial ETA results, combined with the expected gains from the AIO programming model, provide us with the intuition that the combined benefits will yield significantly improved scaling over existing Linux network implementations. To support this intuition, we will begin by measuring simple micro-benchmarks in order to get a baseline performance of the prototype. We will write simple tests based on the TTCP [31] micro-benchmark using the AIO programming model in order to get baseline throughput and round-trip latency performance measurements. We will then quickly move to more realistic workloads. A stretch goal will be to apply other workloads, for example storage or trans-

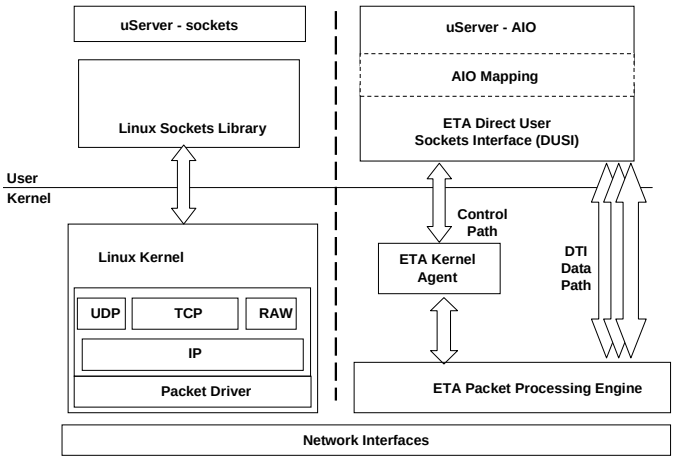

Fig. 6. Proposed Stack

action processing, in order to understand the general efficacy of our approach beyond web serving. In the analysis phase, we will use existing analysis tools, such as VTune [32] and/or oprofile [33], to analyze the impact of our approach at the micro-architectural level. For example, we will gather data on CPU, memory and cache usage. In addition, we will analyze the effect of reduced memory copies, device interrupts and context switches on overall application performance.

Once we have run and analyzed our prototype, we can compare it with other approaches. First we will compare our prototype with a Linux distribution that includes existing network acceleration features such as checksum offload, TCP segmentation offload, and interrupt moderation [34]. Specifically, this comparison will compare our userver application for two cases. The first case is the on standard Linux distribution using standard sockets. The second case is userver running over the DUSI interface to ETA. Figure 6 shows a side-by-side view of the standard Linux networking stack on the left, and our scalable networking prototype stack on the right. This comparison will allow us to measure and analyze the combined benefits of the ETA architecture and a non-blocking, asynchronous application interface.

Given substantial industry activity developing TCP Offload Engine (TOE) devices [13], we will also qualitatively compare our approach to a TOE-based solution running a web-server application.

VII. Summary

In this paper, we have proposed a technology strategy that combines the ETA architecture and the asynchronous I/O programming model to achieve scalable networking on industry standard computing platforms. Our approach potentially enables TCP and emerging high-speed commodity interconnects such as 10 gigabit Ethernet to approach the performance levels of today's SANs. Moreover, our proposed architecture can be a key enabler for a cost-effective unified I/O fabric that supports networking, storage, and inter-process communication.

Our initial experimental results using the existing ETA architecture prototype for kernel-level applications are very promising. Our ongoing work is to complete a prototype of the combined architecture with ETA and the AIO programming model, restructure our user-level web server application to use this platform, and evaluate the performance benefits delivered to the application.

References

[1] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *19th ACM Symposium on Operating Systems Principles*, 2003.

[2] 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.

[3] 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*, vol. 18, no. 2, pp. 66–76, March-April 1998.

[4] InfiniBand^{SM} Trade Association, *InfiniBand^{TM} Architecture Specification Volume 1, Release 1.0*, October 2000. [Online]. Available: www.infinibandta.org

[5] "RDMA Consortium." [Online]. Available: www.rdmaconsortium.org

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

[7] "Design notes on asynchronous I/O (aio) for Linux," 2002. [Online]. Available: lse.sourceforge.net/io/aionotes.txt

[8] "Sockets API Extensions." [Online]. Available: www.opengroup.org

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

[10] "Direct Access File System (DAFS)." [Online]. Available: www.dafscollaborative.org

[11] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen, "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, Aug 1997.

[12] W. Magro, P. Peterson, and S. Shar, "Hyper-threading technology: impact on compute-intensive workloads," *Intel Technology Journal*, Feb 2002.

[13] 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.

[14] 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.

[15] T. Brecht and M. Ostrowski, "Exploring the performance of select-based internet servers," HP Labs, Tech. Rep. HPL-2001-314, November 2001.

[16] HP Labs, "The userver home page," 2003. [Online]. Available: www.hpl.hp.com/research/linux/userver

[17] L. Gwennap, "Count on TCP offload engines," *EE Times*, Sep 2001. [Online]. Available: www.eetimes.com/semi/c/ip/OEG20010917S0051

[18] 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.

[19] P. Buonadonna and D. Culler, "Queue-Pair IP: A hybrid architecture for System Area Networks," in *29th Annual Int'l Symposium on Computer Architecture (ISCA)*, May 2002.

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

[21] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O support in Linux 2.5," in *Proc. of the Linux Symposium*, Jul 2003. [Online]. Available: archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Pulavarty-OLS2003.pdf

[22] 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.

[23] 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. [Online]. Available: citeseer.nj.nec.com/article/pai99flash.html

[24] "Zeus Technology." [Online]. Available: www.zeus.co.uk

[25] 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. [Online]. Available: citeseer.nj.nec.com/586017.html

[26] "Apache." [Online]. Available: www.apache.org

[27] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea for high-concurrency servers," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[28] P. Bonnet, "Personal communication," October 2003.

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

[30] *SPECweb99 Benchmark*, Standard Performance Evaluation Corporation, 1999, http://www.spec.org/osg/web99.

[31] "Test TCP." [Online]. Available: http://www.netcordia.com/tools/tools/TTCP/ttcp.html

[32] "VTune™ Performance Analyzer." [Online]. Available: www.intel.com/software/products/vtune/vpa/index.htm

[33] "OProfile." [Online]. Available: oprofile.sourceforge.net/news/

[34] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet," in *5th Annual Linux Showcase (ALS) and Conference*. USENIX, Nov 2001, pp. 165–172.