Server Network Scalability and TCP Offload

Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz,
Erich Nahum, Prashant Pradhan, John Tracey
IBM T. J. Watson Research Center
Hawthorne, NY, 10532
{dmfreim,elbert,lavoie,mraz,nahum,ppradhan,traceyj}@us.ibm.com

Abstract

Server network performance is increasingly dominated by poorly scaling operations such as I/O bus crossings, cache misses and interrupts. Their overhead prevents performance from scaling even with increased CPU, link or I/O bus bandwidths. These operations can be reduced by redesigning the host/adapter interface to exploit additional processing on the adapter. Offloading processing to the adapter is beneficial not only because it allows more cycles to be applied but also of the changes it enables in the host/adapter interface. As opposed to other approaches such as RDMA, TCP offload provides benefits without requiring changes to either the transport protocol or API.

We have designed a new host/adapter interface that exploits offloaded processing to reduce poorly scaling operations. We have implemented a prototype of the design including both host and adapter software components. Experimental evaluation with simple network benchmarks indicates our design significantly reduces I/O bus crossings and holds promise to reduce other poorly scaling operations as well.

1 Introduction

Server network throughput is not scaling with CPU speeds. Various studies have reported CPU scaling factors of 43% [23], 60% [15], and 33% to 68% [22] which fall short of an ideal scaling of 100%. In this paper, we show that even increasing CPU speeds and link and bus bandwidths does not generate a commensurate increase in server network throughput. This lack of scalability points to an increasing tendency for server network throughput to become the key bottleneck limiting system performance. It motivates the need for an alternative design with better scalability.

Server network scalability is limited by operations heavily used in current designs that themselves do not scale well, most notably bus crossings, cache misses and interrupts. Any significant improvement in scalability must reduce these operations. Given that the problem is one of scalability and not simply performance, it will not be solved by faster processors. Faster processors merely expend more cycles on poorly scaling operations.

Research in server network performance over the years has yielded significant improvements including: integrated checksum and copy, checksum offload, copy avoidance, interrupt coalescing, fast path protocol processing, efficient state lookup, efficient timer management and segmentation offload, a.k.a. large send. Another technique, full TCP offload, has been pursued for many years. Work on offload has generated both promising and less than compelling results [1, 38, 40, 42]. Good performance data and analysis on offload is scarce.

Many improvements in server scalability were described more than fifteen years ago by Clark et al. [9]. The authors demonstrated that the overhead incurred by network protocol processing, per se, is small compared to both per-byte (memory access) costs and operating system overhead, such as buffer and timer management. This motivated work to reduce or eliminate data touching operations, such as copies, and to improve the efficiency of operating system services heavily used by the network stack. Later work [19] showed that overhead of non-data touching operations is, in fact, significant for real workloads, which tend to feature a preponderance of small messages. Today, per-byte overhead has been greatly reduced through checksum offload and zero-copy send. This leaves per-packet overhead, operating system services and zero-copy receive as the main remaining areas for further improvement.

Nearly all of the enhancements described by Clark et al. have seen widespread adoption. The one notable exception is "an efficient network interface." This is a network adapter with a fast general-purpose processor that provides a much more efficient interface to the network than the current frame-based interface devised decades ago. In this paper, we describe an effort to develop a much more efficient network interface and to make this enhancement a reality as well.

Our work is pursued in the context of TCP for three reasons: 1) TCP's enormous installed base, 2) the methodology employed with TCP will transfer to other protocols, and 3) the expectation that key new architectural features, such as zero copy receive, will ultimately demonstrate their viability with TCP.

The work described here is part of a larger effort to improve server network scalability. We began by analyzing server network performance and recognizing, as others have, a significant scalability problem. Next, we identified specific operations to be the cause, specifically: bus crossings, cache misses, and interrupts. We formulated a design that reduces the impact of these operations. This design exploits additional processing at the network adapter, i.e. offload, to improve the efficiency of the host/adapter interface which is our primary focus. We have implemented a prototype of the new design which consists of host and adapter software components and have analyzed the impact of the new design on bus crossings. Our findings indicate that offload can substantially decrease bus crossings and holds promise to reduce other scalability limiting operations such as cache misses. Ultimately, we intend to evaluate the design in a cycle-accurate hardware simulator. This will allow us to comprehensively quantify the impact of design alternatives on cache misses, interrupts and overall performance over several generations of hardware.

This paper is organized as follows. Section 2 provides motivation and background. Section 3 presents our design, and the current prototype implementation is described in Section 4. Section 5 presents our experimental infrastructure and results. Section 6 surveys and contrasts related work, and Section 7 summarizes our contributions and plans for future work.

2 Motivation and Background

To provide the proper motivation and background for our work, we first describe the current best practices of techniques and optimizations for network server performance. Using industry standard benchmarks we then show that, despite these practices, servers are still not scaling with CPU speeds via several benchmarks. Since TCP offload has been a controversial topic in the research community, we review the critiques of offload, providing counterarguments to each point. How TCP offload addresses these scaling issues is described in more detail in Section 3.

2.1 Current Best Practices

Current high-performance servers have adopted many techniques to maximize performance. We provide a brief overview of them here.

Sendfile with zero copy. Most operating systems have a sendfile or transmitfile operation that allows sending a file over a socket without copying the contents of the file into user space. This can have substantial performance benefits [30]. However, the benefits are limited to send-side processing; it does not affect receive-side processing. In addition, it requires the server application to maintain its data in the kernel, which may not be feasible for systems such as application servers, which generate content dynamically.

Checksum offload. Researchers have shown that calculating the IP checksum over the body of the data can be expensive [19]. Most high-performance adapters have the ability to perform the IP checksum over both the contents of the data and the TCP/IP headers. This removes an expensive data-touching operation on both send and receive. However, adapter-level checksums will not catch errors introduced by transferring data over the I/O bus, which has led some to advocate caution with checksum offload [41].

Interrupt coalescing. Researchers have shown that interrupts are costly, and generating an interrupt for each packet arrival can severely throttle a system [28]. In response, adapter vendors have enabled the ability to delay interrupts by a certain amount of time or number of packets in an effort to batch packets per interrupt and amortize the costs [14]. While effective, it can be difficult to determine the proper trigger thresholds for firing interrupts, and large amounts of batching may cause unacceptable latency for an individual connection.

Large send/segmentation offload. TCP/IP implementers have long known that larger MTU sizes provide greater efficiency, both in terms of network utilization (fewer headers per byte transferred) and in terms of host CPU utilization (fewer per-packet operations incurred per byte sent or received). Unfortunately, larger MTU sizes are not usually available due to Ethernet's 1516 byte frame size. Gigabit Ethernet provides "jumbo frames" of 9 KB, but these are only useful in specialized local environments and cannot be preserved across the wide-area Internet. As an approximation, certain operating systems, such as AIX and Linux, provide large send or TCP segmentation offload (TSO) where the TCP/IP stack interacts with the network device as if it had a large MTU size. The device in turn segments the larger buffers into 1516-byte Ethernet frames and adjusts the TCP sequence numbers and checksums accordingly. However, this technique is also limited to send-side processing. In addition, as we demonstrate in Section 2.2, the technique is limited by the way TCP performs congestion control.

Efficient connection management. Early networked servers did not handle large numbers of TCP connections efficiently, for example by using a linear linked-list to manage state [26]. This led to operating systems using hash table based approaches [24] and separating table entries in the TIME_WAIT state [2].

Asynchronous interfaces. To maximize concurrency, high-performance servers use asynchronous interfaces as not to block on long-latency operations [33]. Server applications interact using an event notification interface such as `select()` or `poll()`, which in turn can have performance implications [5]. Unfortunately,

Machine	BIOS Release Date	Clock Speed (MHz)	Cycle Time (ns)	Bus Width (bits)	Bus Speed (MHz)	L1 Size (KB)	L2 Size (KB)	E1000 NICs (num)
Workstation-Class								
500 MHz P3	Jul 2000	500	2.000	32	33	32	512	1
933 MHz P3	Mar 2001	933	1.070	32	33	32	256	1
1.7 GHz P4	Sep 2003	1700	0.590	64	66	8	256	2
Server-Class								
450 MHz P2-Xeon	Jan 2000	450	2.200	64	33	32	2048	2
1.6 GHz P4-Xeon	Oct 2001	1600	0.625	64	100	8	256	3
3.2 GHz P4-Xeon	May 2004	3200	0.290	64	133	8	512	4

Table 1: Properties for Multiple Generations of Machines

these interfaces are typically only for network I/O and not file I/O, so they are not as general as they could be.

In-kernel implementations. Context switches, data copies, and system calls can be avoided altogether by implementing the server completely in kernel space [17, 18]. While this provides the best performance, in-kernel implementations are difficult to implement and maintain, and the approach is hard to generalize across multiple applications.

RDMA. Others have also noticed these scaling problems, particularly with respect to data copying, and have offered RDMA as a solution. Interest in RDMA and Infiniband [4] is growing in the local-area case, such as in storage networks or cluster-based supercomputing. However, RDMA requires modifications to both sides of a conversation, whereas Offload can be deployed incrementally on the server side only. Our interest is in supporting existing applications in an inter-operable way, which precludes using RDMA.

While effective, these optimizations are limited in that they do not address the full range of scenarios seen by a server. The main restrictions are: 1) that they do not apply to the receive side, 2) they are not fully asynchronous in the way they interact with the operating system, 3) they do not minimize the interaction with the network interface, or 4) they are not inter-operable. Additionally, many techniques do not address what we believe to be the fundamental performance issue, which is overall server scalability.

2.2 Server Scalability

The recent arrival of 10 gigabit Ethernet and the promise of 40 and 100 gigabit Ethernet in the near future show that raw network bandwidth is scaling at least as quickly as CPU speed. However, it is well-known that memory speeds are not scaling as quickly as CPU speed increases [16]. As a consequence of this and other factors, researchers have observed that the performance of host TCP/IP implementations is not scaling at the same rate as CPU speeds in spite of raw network bandwidth increases.

To quantify how performance scales over time, we ran a number of experiments using several generations of machines, described in detail in Table 1. We break the machines into 2 classes: desk-side workstations and rack-mounted servers with aggressive memory systems and I/O busses. The workstations include a a 500 MHz Intel Pentium 3, a 933 MHz Intel Pentium 3, and a a 1.7 GHz Pentium 4. The servers include a 450 MHz Pentium II-Xeon, a 1.6 GHz P4 Xeon, and a 3.2 GHz P4 Xeon. In addition, each of the P4-Xeon servers have 1 MB L3 caches. Each machine runs Linux 2.6.9 and has a number of Intel E1000 MT server gigabit Ethernet adapters, connected via a Dell gigabit switch. Load is generated by five 3.2 GHz P4-Xeons acting as clients, each using an E1000 client gigabit adapter and running Linux 2.6.5. We chose the E1000 MT adapters for the servers since these have been shown to be one of the highest-performing conventional adapters on the market [32], and we did not have access to a 10 gigabit adapter.

We measured the time to access various locations in the memory hierarchy for these machines, including from the L1 and L2 caches, main memory, and the memory-mapped I/O registers on the E1000. Memory hierarchy times were measured using LMBench [25]. To measure the device I/O register times, we added some modifications to the initialization routine of the Linux 2.6.9 E1000 device driver code. Table 2 presents the results. Note that while L1 and L2 access times remain relatively consistent in terms of processor cycles, the time to access main memory and the device registers is increasing over time. If access times were improving at the same rate as CPU speeds, the number of clock cycles would remain constant.

To see how actual server performance is scaling over time, we ran the static portion of SPECweb99 [12] us-

Machine	L1 Cache Hit		L2 Cache Hit		Main Memory		I/O Register Read		I/O Register Write	
	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles	Time (ns)	Clock Cycles
Workstation-Class										
500 MHz P3	6	3	44	22	162	80	600	300	300	150
933 MHz P3	3.25	3	7.5	7	173	161	700	654	400	373
1.7 GHz P4	1.2	2	10.9	18	190	323	800	1355	100	169
Server-Class										
450 MHz P2-Xeon	6.75	3	38.3	17	207	93	800	363	200	90
1.6 GHz Xeon	1.37	2	11.57	18	197	315	900	1440	300	480
3.2 GHz Xeon	0.6	2	5.8	18	111	376	500	1724	200	668

Table 2: Memory Access Times for Multiple Generations of Machines

ing a recent version of Flash [33, 37]. In these experiments, Flash exploits all the available performance optimizations on Linux, including `sendfile()` with zero copy, TSO, and checksum offload on the E1000. Table 3 shows the results. Observe that server performance is not scaling with CPU speed, even though this is a heavily optimized server making use of all current best practices. This is not because of limitations in the network bandwidth; for example, the 3.2 GHz Xeon-based machine has 4 gigabit interfaces and multiple 10 gigabit PCI-X busses.

2.3 Offload: Critiques and Responses

In this paper, we study TCP offload as a solution to the scalability problem. However, TCP offload has been hotly debated by the research community, perhaps best exemplified by Mogul's paper, "TCP offload is a dumb idea whose time has come" [27]. That paper effectively summarizes the criticisms of TCP offload, and so, we use the structure of that paper to offer our counterarguments here.

Limited processing requirements. One argument is that Clark et al. [9] show that the main issue in TCP performance is implementation, not the TCP protocol itself, and a major factor is data movement; thus Offload does not address the real problem. We point out that Offload does not simply mean TCP header processing; it includes the entire TCP/IP stack, including poorly-scaling, performance-critical components such as data movement, bus crossings, interrupts, and device interaction. Offload provides an improved interface to the adapter that reduces the use of these scalability-limiting operations.

Moore's Law: Moore's Law states that CPU speeds are doubling every 18 months, and thus one claim is that Offload cannot compete with general-purpose CPUs. Historically, chips used by adapter vendors have not increased at the same rate as general-purpose CPUs due to the economies of scale. However, offload can use commodity CPUs with software implementations, which we believe is the proper approach. In addition, speed needs only to be matched with the interface (e.g., 10 gigabit Ethernet), and we argue proper design reduces the code path relative to the non-offloaded case (e.g. with fewer memory copies). Sarkar et al. [38] and Ang [1] show that when the NIC CPU is under-provisioned with respect to the host CPU, performance can actually degrade. Clearly the NIC processing capacity must be sized properly. Finally, increasing CPU speeds does not address the scalability issue, which is what we focus on here.

Efficient host interface: Early critiques are that TCP Offload Engines (TOE) vendors recreated "TCP over a bus". Development of an elegant and efficient host/adapter interface for offload is a fundamental research challenge, one we are addressing in this paper.

Bad buffer management: Unless Offload engines understand higher-level protocols, there is still an application-layer header copy. While true, copying of application headers is not as performance-critical as copying application data. One complication is the application combining its own headers on the same connection with its data. This can only be solved by changing the application, which is already proposed in RDMA extensions for NFS and iSCSI [7, 8].

Connection management overhead: Unlike conventional NICs, offload adapters must maintain per-connection state. Opponents argue that offload cannot handle large numbers of connections, but Web server workloads have forced host TCP stacks to discover techniques to efficiently manage 10,000's of connections. These techniques are equally applicable for an interface-based implementation.

Resource management overhead: Critics argue that tracking resource management is "more difficult" for offload. We do not believe this is the case. It is straight-

Machine	Throughput (ops/sec)	Requested Connections	Conforming Connections	Scale (achieved)	Scale (ideal)	Ratio (%)
Workstation-Class						
500 MHz P3	1231	375	375	1.00	1.00	100
933 MHz P3	1318	400	399	1.06	1.87	56
1.7 GHz P4	3457	1200	1169	3.20	3.40	94
Server-Class						
450 MHz P2-Xeon	2230	700	699	1.00	1.00	100
1.6 GHz P4-Xeon	8893	2800	2792	4.00	3.56	112
3.2 GHz P4-Xeon	11614	2500	3490	5.00	7.10	71

Table 3: SPECWeb99 Performance Scalability over Multiple Generations of Machines

forward to extend the notion of resource management across the interface without making the adapter aware of every process as we will show in Sections 3 and 4.

Event management: The claim is that offload does not address managing the large numbers of events that occur in high-volume servers. It is true that offload, per se, does not address *application visible* events, which are better addressed by the API. However, offload can shield *the host operating system* from spurious unnecessary *adapter events*, such as TCP acknowledgments or window advertisements. In addition, it allows batching of other events to amortize the cost of interrupts and bus crossings.

Partial offload is sufficiently effective: Partial offload approaches include checksum offload and large send (or TCP Segmentation Offload), as discussed in Section 2.1. While useful, they have limited value and do not fully solve the scalability problem as was shown in Section 2.2. Other arguments include that checksum offload actually masks errors to the host [41]. In contrast, offload allows larger batching and the opportunity to perform more rigorous error checking (by including the CRC in the data descriptors).

Maintainability: Opponents argue that offload-based approaches are more difficult to update and maintain in the presence of security and bug patches. While this is true of an ASIC-based approach, it is not true of a software-based approach using general-purpose hardware.

Quality assurance: The argument here is that offload is harder to test to determine bugs. However, testing tools such as TBIT [31] and ANVL [11] allow remote testing of the offload interface. In addition, software based approaches based on open-source TCP implementations such as Linux or FreeBSD facilitate both maintainability and quality assurance.

System management interface: Opponents claim that offload adapters cannot have the same management interface as the host OS. This is incorrect: one example is SNMP. It is trivial to extend this to an offload adapter.

Concerns about NIC vendors: Third-party vendors may go out of business and strand the customer. This has nothing to do with offload; it is true of any I/O device: disk, NIC, or graphics card. Economic incentives seem to address customer needs. In addition, one of the largest NIC vendors is Intel.

3 System Design

In this Section we describe our Offload design and how it addresses scalability.

3.1 How Offload Addresses Scalability

A higher-level interface. Offload allows the host operating system to interact with the device at a higher level of abstraction. Rather than simply queuing MTU-sized packets for transmission or reception, the host issues commands at the transport layer (e.g., `connect()`, `accept()`, `send()`, `close()`). This allows the adapter to shield the host from transport layer events (and their attendant interrupt costs) that may be of no interest to the host, such as arrivals of TCP acknowledgments or window updates. Instead, the host is only notified of meaningful events. Examples include a completed connection establishment or termination (rather than every packet arrival for the 3-way handshake or 4-way tear-down) or application-level data units. Sufficient intelligence on the adapter can determine the appropriate time to transfer data to the host, either through knowledge of standardized higher-level protocols (such as HTTP or NFS) or through a programmable interface that can provide an application signature (i.e., an application-level equivalent to a packet filter). By interacting at this higher level of abstraction, the host will transfer less data over the bus and incur fewer interrupts and device register accesses.

Ability to move data in larger sizes. As described in Section 2.1, the ability to use large MTUs has a significant impact on performance for both sending and re-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.