Improving Network Processing Concurrency using TCPServers*

Aniruddha Bohra† and Liviu Iftode
Department of Computer Science, Rutgers University,
Piscataway, NJ 08854
{bohra,iftode}@cs.rutgers.edu

Abstract

Exponentially growing bandwidth requirements and slowing gains in processor speeds have led to the popularity of multiprocessor architectures. Network stack parallelism is increasingly important to support such architectures. In this paper, we present techniques to improve network stack concurrency using our previous work, TCPServers, a system architecture for offloading network processing within an SMP system. TCPServers dedicates a subset of processors as packet processing engines (PPEs), which handle all asynchronous network events and perform receive processing. We introduce Receive Queues, data structures associated with each socket that store incoming network packets and are accessed exclusively at the PPEs. Using Receive Queues, we modify TCPServers based network stacks to incorporate early packet demultiplexing. We also present an efficient proportional fair scheduling algorithm, which processes incoming packets at the priority of the destination socket.

Our experimental evaluation demonstrates that our modifications reduce the scheduling and synchronization overheads and improve the aggregate TCP/IP throughput by up to 75% compared against the default SMP stack. We also show that our system sustains this throughput, even when a large number of short lived connections are present.

1. Introduction

While network bandwidths are growing exponentially, processor speeds are growing at a slower rate than ever before. To make up for the loss in performance, CPU vendors have started focusing on increased parallelism. Symmetric Multiprocessing (SMP), hyperthreading or Simultaneous Multithreading (SMT), and multicore or Chip Multiprocessing (CMP) architectures are becoming increasingly common.

Previous studies have indicated that synchronization and scheduling overheads in a multiprocessor system limit the performance supported by a parallel network stack [12, 18]. Shared data structures, for example the global connection table, are protected in a multiprocessor system using expensive synchronization primitives. Moreover, these overheads increase with the number of processors in the system and the number of contending threads. Recently, there has been an effort to improve the scalability of network stacks across multiple processors.

In this paper, we present techniques to improve network stack concurrency using our previous work – TCPServers [14], where a subset of processors in an SMP system is dedicated to network processing. The dedicated packet processing engines (PPEs) handle all asynchronous network events and employ a hybrid polling-interrupt mechanism to prevent receive livelock, which affects interrupt based network stacks [9].

We design a multiprocessor stack to eliminate accesses to globally shared data structures from the critical network processing path, thus reducing synchronization overheads. We introduce Receive Queues, private data structures associated with each socket, which store packets received from the network at the PPE. Receive Queues are fixed sized data structures that are accessed at the PPE on receiving packets, and by a single protocol processing thread. Since there is one producer and one consumer, both residing on a single processor, Receive Queues can be protected using inexpensive atomic operations, for example, test and set.

In TCPServers, we use early demultiplexing, where incoming packets are classified in the interrupt handler and placed in the corresponding Receive Queue. Early demultiplexing combined with lazy scheduling of network processing has previously been shown to enable graceful performance degradation for web-servers when overloaded and fair resource allocation across communicating processes [5]. Unfortunately, process priorities are not available at the PPE. Therefore, fair scheduling of network processing in TCPServers is challenging. We present two variants of our system, TCPS Early and TCPS Sched, which implement early demultiplexing. While TCPS Early enforces a round-robin scheduling discipline, TCPS Sched uses an efficient proportional fair scheduling algorithm.

*This work is supported by the National Science Foundation grant CCR 0133366.

†Also with NEC Labs America

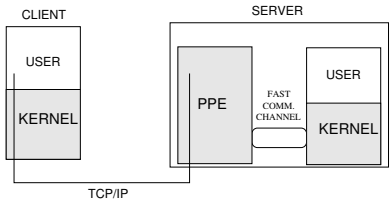

Figure 1. TCPServers Architecture. The Packet Processing Engine handles all network events and communicates with application processors over a fast communication channel. For SMP systems, this channel is shared memory.

We design and implement all TCPServers variants in the FreeBSD operating system and present an evaluation of their performance. Our results demonstrate that the default SMP implementation does not sustain high throughput in presence of a large number of concurrent connections. Through our experiments, we show that with our modifications, TCPServers improves the aggregate throughput by up to 75% of the default SMP implementation. We also demonstrate that with early demultiplexing and priority based scheduling, TCPServers sustains high data transfer rates even in the presence of short-lived connections where the performance of other implementations degrades significantly.

This paper is organized as follows: We present a brief overview of TCPServers in Section 2 and describe the design of our system in Section 3. We discuss our implementation in Section 4 and present an evaluation of our system in Section 5. Section 6 discuss related work and Section 7 concludes the paper.

2. TCPServers

TCPServers [14] (Figure 1) is a system architecture we previously proposed previously for offloading network processing from application hosts to dedicated processors in a multiprocessor environment, nodes in a cluster, or in a cluster of intelligent devices (CID). TCPServers dedicates a subset of processors to handle network processing. We call these processors Packet Processing Engines (PPEs) and the remaining processors are called application processors. For multiprocessor environments, the PPEs and application processors communicate over shared memory. For cluster and CID environments, the PPEs and the application processors communicate over a high-bandwidth, low-latency interconnect, e.g. Infiniband [7].

In a multiprocessor system, which is the focus of this paper, PPEs handle the asynchronous events, e.g., NIC interrupts, network timers, deferred send processing, and receive processing. In the send processing path, PPEs are invoked only if the network processing is deferred, e.g. waiting for an interface queue to be drained, or for ARP processing. Receive processing for the TCP/IP protocol requires asynchronous protocol processing to generate ACKs, to send any pending data in response to received ACKs, and to retransmit packets on loss detection or on timeouts. These tasks cannot be delayed to be performed in the context of the receiving process, therefore PPEs handle the majority of the receive processing. The receiving process context is used only to perform the copy from the kernel to user buffers.

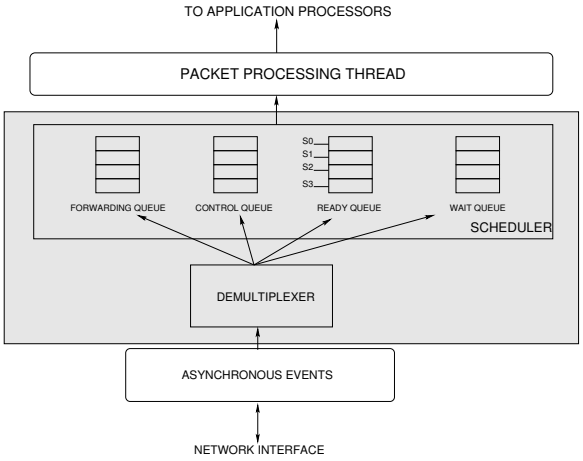

Figure 2. TCPServers software architecture. The shaded region shows extensions presented in this paper.

Figure 2 shows the components of TCPServers software architecture at the PPE. Each PPE has three main components: an asynchronous event handler, a scheduler, and a Packet Processing Thread (PPT). The shaded components in the figure are extensions to our previous work and are described in more detail in Section 3.

TCPServers uses an adaptive polling mechanism for handling NIC generated events. The NIC generates an interrupt on the network activity edge, that is, the first received packet generates an interrupt, which is followed by a sequence of polls to handle received packets. Polling is performed in the software interrupt context (softirq), which executes at the PPE at the highest priority. During polling, the NIC interrupt mechanism is disabled. When polling generates no new events, interrupts are re-enabled at the NIC.

A PPT is a kernel-resident thread that performs network processing and is bound to a PPE. Network receive processing runs to completion and is non-blocking, therefore the PPT never yields the processor. Moreover, the scheduler does not schedule any other process on PPEs, which are exclusively used for network processing.

3. Design

In this section, we describe the design of the TCPServers extensions for early demultiplexing of network packets using Receive Queues and scheduling network processing at the priority of the corresponding application (socket).

The shaded region in Figure 2 shows the TCPServers components introduced in this paper. The scheduler maintains per-socket Receive Queues, classified as forwarding, control, ready, and wait queues, which are used to store packets destined for the corresponding socket. The demultiplexer places incoming packets in the appropriate queue and activates the scheduler, which chooses the next receive queue for processing and invokes the packet processing thread. In the following, we discuss each of the above in more detail.

3.1. Receive Queues

A Receive Queue (RQ) is an OS data structure that is used by TCPServers to enqueue protocol processing requests to PPTs or to application processes. An RQ is a mailbox associated with each socket, where received packets are queued by the asynchronous event handler and are dequeued by the PPT. The RQ is addressed by the connection identifier that identifies a socket.

Receive queues are implemented as fixed sized arrays, which are protected by per-CPU atomic operations, for example test and set or compare and swap operations. Since there is one producer (asynchronous event handler) and one consumer for each queue (PPT), the atomic operation based mutual exclusion is sufficient. This eliminates not only expensive synchronization operations but also the contention on the globally shred IP queue.

Receive queues are associated with a socket when the socket's address is identified on one of the following events: *(i)* Local ports are allocated implicitly or through a `bind()` system call, *(ii)* The OS creates a new socket when a stream based socket is connected and queued for the application process to issue the `accept()` system call, *(iii)* Sockets are bound to the same UDP multicast group, and *(iv)* Globally shared queues are created at startup for packets received for unknown connections or IP fragments that do not contain the transport headers.

3.2. Early Demultiplexing

Traditional networking stacks handle hardware interrupts and queue the received packet in the IP queue for processing in the software interrupt context. TCPServers eliminates the IP queue. Instead, the asynchronous event handler demultiplexes the incoming packets early into per-socket receive queues.

Packet demultiplexing initially uses the Ethernet header to identify the network protocol (ARP, IP, PPP, IPX etc.). IP packets are further classified using the five tuple flow identifier to uniquely identify the connection. Packets are queued into the receive queue if there is an empty slot and are dropped otherwise. IP fragments that do not contain a transport layer header are placed in a shared queue for reassembly.

Through early demultiplexing and fixed sized per-socket receive queues, TCPServers creates a feedback mechanism. Receive queues for applications (sockets) that are inactive or cannot handle the packets at the incoming rate overflow leading to packet loss and lower performance. On the other hand, well-behaved applications that can handle the rate of incoming packets enjoy greater stability under load.

Demultiplexing packets into receive queues increases the overhead of the asynchronous event handler (interrupt handler or softirq handler). However, there are three advantages of early demultiplexing. First, the socket buffer can be processed on application processors while the event handler is executing on the PPE. Second, packets are dropped from unresponsive socket queues without performing expensive IP and transport protocol processing. Third, unlike the shared IP queue, which requires expensive mutual exclusion primitives, the per-socket receive queues are protected using simple and inexpensive atomic operations.

3.3. Scheduling Network Processing

TCPServers requires a mechanism to schedule network protocol processing in the PPT context. Integrating the PPT with the system-wide scheduler defeats the purpose of offloading. Therefore, we need a mechanism independent of the system scheduler that prioritizes network event processing across receive queues. Network processing has several unique characteristics that affect the design of such a scheduling algorithm: *(i)* Scheduling a *network processing task* does not involve a context switch. Instead, it is a function call and therefore does not incur any additional overhead, *(ii)* Network processing in the PPT always runs to completion. The tasks do not block waiting for other tasks or OS resources, e.g. IPC or disk. Therefore, no additional state must be preserved across executions, and *(iii)* Network processing priority is determined not only by the resources spent on servicing the socket, but also by the network *protocol semantics*. For example, IP forwarding packets must be handled at a higher priority than the packets destined for the host; in a busy server, packets that terminate connections (FIN, RST) must be handled as soon as possible to reclaim system resources.

Packet Classes and Scheduling Queues: The PPT must handle packets for IP forwarding, packets received for active sockets (connected TCP and bound UDP sockets), connection requests for new connections, and packets belonging to multiple protocols, e.g. TCP/UDP, ICMP, ARP, etc.

The packet classes above are identified by the asynchronous event handler at the time it performs early demultiplexing into receive queues. We maintain four classes of queues in the decreasing order of priority: *(i)* Forwarding Queue, where receive queues for IP forwarding are maintained, *(ii)* Control Queue, where receive queues for control protocols and closed connections are maintained, *(iii)* Ready Queue, where all receive queues with packets pend-

ing for network stack processing are maintained, and *(iv)* Wait Queue, where the remaining receive queues are maintained.

The packets in a higher priority queue are serviced before moving to the next queue. Except the Ready Queue, packets in all other queues are of identical priority, therefore a simple round robin scheduling mechanism is used to service those queues in the PPT. When more than one processor is used as a packet processing engine, one PPE services all queues, while the other processors handle only the Ready Queue.

The Receive Queues in the Ready Queue are organized further into priority groups based on the responsiveness of the application and the number of pending packets received for the socket. The TCP Server scheduling algorithm tries to ensure each priority group receives its fair share of the PPE CPU resources.

Scheduling Priority: The Receive Queue priority captures the responsiveness of the parent application (socket). Applications that are unresponsive have data waiting in the socket queues for `receive` system calls. Therefore, packets received for these applications can be delayed as there already is data to be delivered to the application if required. On the other hand, applications with pending send buffers require urgent service as these applications are waiting on the network subsystem to send out the bytes already present in the socket buffers. Finally, if there are neither pending sends nor receives, applications that have received more packets require service before those that have received only a few packets.

The scheduling priority or weight that captures the above criteria allows the network process scheduling to ensure prioritized packet processing for active and responsive applications. We define the scheduling priority as a linear combination of the weights assigned to the pending send and pending receive packets at the socket, and the pending packets in the RQ denoted by W_s, W_r, and W_q respectively. These weights are defined for a socket i as

$$\begin{aligned} W_s(i) &= \frac{PendingSendBytes(i)}{SegmentSize(i)} \\ W_r(i) &= \frac{MaxRcvBytes(i) - PendingRcvBytes(i)}{SegmentSize(i)} \\ W_q(i) &= NumPendingPackets(i) \end{aligned} \quad (1)$$

$W_s(i)$ and $W_r(i)$ are approximate number of pending send and receive *packets*. In the above, $MaxRcvBytes(i)$ represents the socket's receive buffer size, and $SegmentSize(i)$ is the size of a segment of data that is sent out over the connection. While this measure is not exact, it approximates the number of packets pending attention and the number of packets that can be served for this socket.

Unfortunately, the equations above ignore the applications that set a very large receive socket buffer. In that scenario, $MaxRcvBytes(i)$ dominates $W_r(i)$ and cannot be easily compared against other sockets. To overcome this limitation, we normalize the weights and redefine the weights as

$$\begin{aligned} \widehat{W}_s(i) &= \frac{W_s(i) \times MaxSndBytes}{MaxSndBytes(i)} \\ \widehat{W}_r(i) &= \frac{W_r(i) \times MaxRcvBytes}{MaxRcvBytes(i)} \\ \widehat{W}_q(i) &= \frac{W_q(i) \times MaxRQSize}{MaxRQSize(i)} \end{aligned} \quad (2)$$

To avoid fractional weights, we scale the normalized weights by the maximum allowed send and receive buffer sizes, and redefine the scheduling weight as

$$W(i) = \widehat{W}_s(i) + \widehat{W}_r(i) + \widehat{W}_q(i) \quad (3)$$

Scheduling Algorithm: TCPServers uses a proportional share scheduling algorithm derived from the recently proposed Group Ratio Round-Robin [4] CPU scheduling algorithm. The GR^3 algorithm provides constant fairness bounds on proportional sharing accuracy with O(1) scheduling overhead. The key idea is to define groups of RQs (clients) with closely related priorities and keep the groups in sorted order, while keeping the RQs (clients) within the group in an unsorted list. Scheduling across groups uses the ratio of group weights (sum of weights of all members of the group) to determine which group to select. Since the groups are kept in a sorted list, choosing a group requires only a constant time comparison of two group weights. Within the group, RQs are scheduled using a simple Deficit Round Robin algorithm, where the RQ is scheduled for time slots proportional to its normalized weight within the group. Fractional shares are deferred for the next round, where the *deficit* is added to the client's weight.

Figure 3 shows the algorithm used by TCPServers to schedule network processing at the PPT. ϕ is the weight assigned to each group G, while D is the deficit maintained within a group to perform a deficit round robin scheduling. $curpos$ is the current position in the group. The $TCPServerSchedule$ subroutine picks the next group to schedule (G_i) and passes a pointer to this group to the $TCPServerGroupSchedule$, which returns the socket whose packets are processed.

Groups are defined for RQs with closely related weights. The *closeness* in weights is defined as a logarithmic order, where a group of order σ, contains all clients whose weights ($W(i)$) follow

$$2^{\sigma} \leq W(i) \leq 2^{\sigma+1} - 1 \quad (4)$$

```
TCPSERVERSCHEDULE()
1  G_i ← Groups[i]
2  S ← TCPServerGroupSchedule(G_i)
3  W_i ← W_i + 1
4  if i < g and (W_i+1)/(W_i+1+1) > φ_i/φ_i+1
5     then i ← i + 1
6     else i ← 1
7  return S

TCPSERVERGROUPSCHEDULE(G)
1  C ← G[curpos]
2  if D_C < 1
3     then curpos ← curpos + 1
4          C ← G[curpos]
5          D_C ← D_C + φ_C/φ^G_min
6  D_C ← D_C − 1
7  return C
```

Figure 3. Scheduling Algorithm for TCPS Sched

Using the logarithmic relationship reduces the number of groups and therefore the maintenance overheads for the sorted group list. Moreover, all clients within a group have their priorities within a factor of two.

Unlike in GR^3, the RQ weights change frequently, on receiving a packet and on sending packets. TCPServers recalculates the priority for all RQs on which packets are received before returning from the asynchronous event handler. The RQs are removed from their current group and are re-inserted into the group which satisfies Equation 4. Removal and insertion of an RQ is an $O(g)$ operation, where g is the number of groups in the system since a linear scan is required to identify the group for the modified RQ. However, even for large 32 bit weights, there are a small number (32) of queues in the system and this scan is not too expensive.

4. Implementation

We have implemented TCPServers in the FreeBSD 7.0 OS. We use the fast interrupt handler for handling the network interrupts instead of the default interrupt thread based implementation. a deferred task-queue We implement three versions of the TCPServers, *(i)* TCPServers, where the PPT is bound to one processor of the multiprocessor, *(ii)* TCPS Early, where we implement early demultiplexing through Receive Queues in addition to the base TCP Server implementation, and *(iii)* TCPS Sched, where we implement our socket aware scheduling algorithm in addition to the TCPS Early implementation.

We modify the system scheduler to exclude the PPE from its scheduling discipline and bind the protocol processing thread to it. This eliminates all other processing from the PPE and dedicates it to network processing. Since the protocol processing thread does not block and does not yield, we do not have any idle time on the PPE.

5. Evaluation

We perform all experiments using an SMP Dell Poweredge 2600 server with two 2.8 GHz Intel Xeon processors, each with 512KB L2 cache. The system has 3GB RAM, and two Intel 82544 gigabit Ethernet adapters connected over a 66MHz/64bit PCI-X bus interface. We use two identical clients connected back-to-back with the server interfaces using cross-over cables. In all our experiments, the client systems are not overloaded and the performance is determined solely by the server system. The server runs a modified version of the FreeBSD-Current as of September 20, 2006, while the clients run Linux Fedora Core 3 with a 2.6.18 kernel.

Benchmark: Most existing network performance benchmarks are not designed for a large number of simultaneous connections and typically measure the peak bandwidth using a single connection. To overcome this limitation, we implemented a benchmark program using the libevent event driven programming library [13]. The benchmark operates in two phases. During the startup phase, it establishes the requested number of connections and in the measurement phase, the server sends data to the client over each of the connections. The data is obtained from an in-memory file using the zero copy `sendfile` interface. We set the send and receive buffers for all sockets as 64KB, as this yields the maximum performance in our experiments. The benchmark runs for 3 minutes (180 seconds) and the data is continuously transferred for the duration of the test in fixed sized blocks of 8KB. The clients discard all received data and send an acknowledgement at the end of each block. On receiving the acknowledgement, the server queues the next block for transfer to the socket. For fine-grained measurements, we use the hardware performance monitoring counters exported by the `hwpmc` driver in the FreeBSD kernel.

5.1. Network Stack Characterization

We first characterize the network stack performance using our synthetic benchmark. Figure 4 shows the aggregate throughput observed across all connections, as the number of connections are increased for the uniprocessor (UP) and a multiprocessor (SMP) kernel. We observe that, while the UP performance degrades as the number of connections increases, the SMP performance initially increases and then degrades. The initial increase is due to the additional parallelism being exploited by the multiprocessor kernel.

The performance of both systems is much lower than the theoretical maximum. A breakdown of the CPU utilization is shown for the SMP kernel in Figure 5. We show the idle time, time spent in the scheduler and synchronization, the network stack, and the user-level processing. We observe that as the number of connections increases, the synchronization and scheduling overhead increases significantly, reducing the time spent in the network stack. This

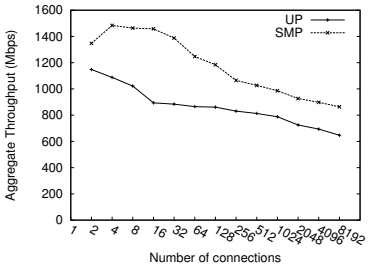

Figure 4. Aggregate throughput.

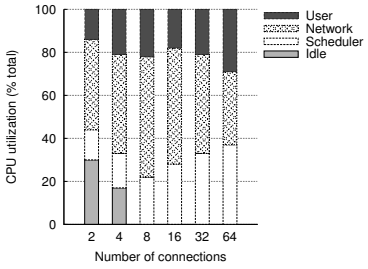

Figure 5. CPU Utilization for default SMP.

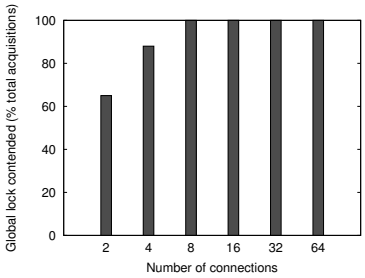

Figure 6. Lock contention in default SMP.

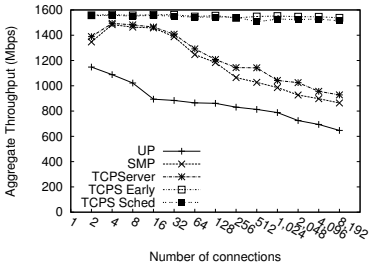

Figure 7. Throughput with increasing number of connections for different network stack organizations.

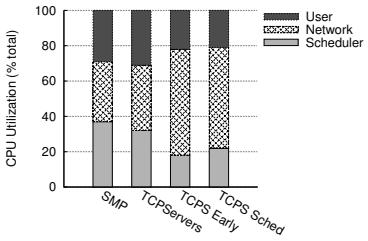

Figure 8. CPU utilization and breakdown for 256 concurrent connections.

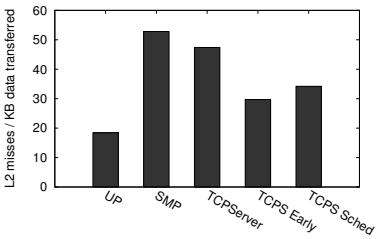

Figure 9. Number of L2 cache misses per KB of data transferred with 256 concurrent connections

leads to a drop in performance as the number of connections increases.

To further analyze the performance, in Figure 6 we show the effects of lock contention. In a multiprocessor system, a contended lock leads to the executing thread being suspended and woken up later when the current owner releases the lock. Therefore, contended locks lead to a higher synchronization, as well as, context switch overhead. To illustrate the effect, we focus on the global connection table lock, which is acquired on every send and receive operation to locate the corresponding connection structure from the list of all connections. We see that the contention for the global lock increases and beyond 8 simultaneous connections, the lock is always contended. From the above, we conclude that in order to improve network performance, we must reduce the lock contention as well as scheduler overheads from the network processing.

5.2. Performance

Figure 7 shows the performance of three variants of TCPServers compared against the default SMP performance. The uniprocessor (UP) performance is included as a baseline reference. We observe that for up to 16 connections, the SMP performance is comparable to the base TCPServers. This variant of TCPServers reduces the scheduler overheads and dedicates one processor in a 2-way SMP server to network processing. However, it still suffers from the higher lock contention overheads, which start dominating at higher connection loads. The TCPS Early and TCPS Sched eliminate the lock contention since they do not access the global connection structure and access only the receive and send queues, which are protected by atomic operations. Therefore, these variants provide a steady throughput even in the presence of a large number of concurrent connections.

The higher throughput of the TCPServers variants is also reflected in the CPU utilization showing the fraction of CPU used by the scheduler and synchronization, the network stack, and the user-level processing. Figure 8 shows the CPU utilization for 256 concurrent connections. Here, the SMP system suffers from high scheduler overheads and therefore the network stack has a smaller fraction and cannot handle the high volume of traffic. In contrast, all variants of TCPServers provide the network stack with a higher fraction of CPU time by dedicating a processor. A lower synchronization overhead further explains the higher performance of the TCPS Early and TCPS Sched.

Finally, to illustrate the effects of L2 cache misses due to data migration, we show the number of L2 cache misses per KB of data transferred for 256 concurrent connections in Figure 9. We observe that the the L2 misses for the SMP and TCPServers variants is higher than the TCPS Early and TCPS Sched since there is limited data migration. Among TCPS Early and TCPS Sched, the scheduling data structures lead to a higher L2 cache miss rate and thus a slight

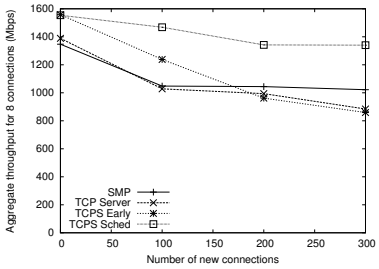

Figure 10. Aggregate throughput with 8 connections with varying number of short-lived connections

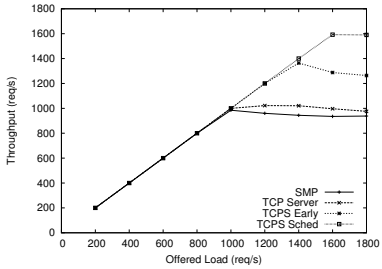

Figure 11. Web server throughput with varying load across network stack variants

performance degradation.

To illustrate the benefit of scheduling network processing at the priority of the destination socket, we first create 8 concurrent connections. We then run a separate process where clients create a number of short-lived connections. These connections are closed as soon as the user-level program receives a completed connection notification. Figure 10 shows the aggregate throughput as the number of short-lived connections increases. Since the new connection establishment leads to higher overhead and leads to packets belonging to other (active) connections being dropped, the throughput decreases. In contrast, the TCPS Sched assigns a lower priority to the new connections (SYN), and leads to a lower degradation in the performance.

5.3. Web Server Performance

To evaluate the impact the our system on a real-world application, we measure the performance of a web-server using each of the variants of our system. We use the Apache 2.0.48 [1] web server and the httperf [10] benchmark to characterize the performance. Httperf maintains a fixed request rate defined in the configuration and measures the number of completed requests within a specified timeout during a run. We generate a trace where all requests are static, for 32KB web pages. Our goal is to measure the performance improvement due to increased concurrency, therefore, we design a workload mix whose working set fits in the available memory. To avoid noise from the disk effects, we discard the results from the first run of our benchmark for all cases.

Figure 11 shows the performance of the web server measured by httperf while varying the offered load. The performance measure is the number of successful requests. We observe that for underloaded conditions, all systems demonstrate similar performance. However, as the request load increases, the TCPS Sched and TCPS Early can support a higher request load. Finally, TCPS Sched sustains the high load and shows graceful degradation even when overloaded. The gap between TCPS Early and TCPS Sched when overloaded is due to the large number of new connections being established continuously. While TCPS Early handles all packets at the same priority, TCPS Sched handles the data packets (requests from established sessions) before handling new connections. This allows more HTTP requests to be satisfied, better utilizing the CPU.

6. Related Work

Interrupt Mitigation: Interrupt processing imposes significant overheads on busy network servers. Interrupts affect the system performance directly as interrupt processing executes at the highest priority and prevents any other processing in the system. Mogul and Ramakrishnan [9] demonstrate that in a loaded system, interrupt processing can easily overwhelm the server CPU leading to *receive livelock*. Interrupt moderation through interrupt batching, polling, and hybrid interrupt and polling architectures have been proposed to mitigate receive livelock in busy systems [16]. Unlike the above, TCP Servers dedicates a subset of processors in a multiprocessor system to handle the network processing to alleviate cache and TLB pollution. It also eliminates globally shared data structures to reduce the synchronization and scheduler overheads.

Lazy Receiver Processing (LRP) [5] uses early demultiplexing and defers packet processing to execute in the context of the process that owns the connection. LRP integrates network processing into the system's global resource management, schedules network processing at the priority of the receiving process, and discards packets of unresponsive processes early in order to conserve system resources and limit the effects of unfair resource allocation. While LRP techniques focus on efficient and fair resource allocation, TCP servers provides separation of application and network functionality.

Network Stack Parallelization: There is a large body of work which studies the parallelization of network stacks [19, 2, 17]. Willmann et. al. perform a comparative study of message and connection-oriented parallelism on modern hardware and conclude that while the connection oriented parallelism offers benefits over message oriented parallelism, the scheduling overheads significantly reduce the efficiency [18]. In this paper, we focus on offloading network functionality to a set of processors in a multipro-

cessor system. This organization improves cache locality and reduces lock contention, which are identified as two important sources of inefficiency in the above.

Offloading: TCP/IP offload engines (TOEs) propose an aggressive offloading strategy and offload stateful tasks, e.g., TCP/IP state maintenance, IP fragment reassembly, TCP reordering, global connection variable maintenance, etc., to the NIC. Such offloading is complex and requires significant CPU and memory resources at the NIC. A hybrid approach to offloading, where the connection offload is controlled and managed by the OS has been proposed recently [8, 6]. These systems target the poorly scaling operations like I/O bus crossings, cache misses, and interrupt processing. While such approaches demonstrate a significant performance gain for a reasonable number of connections, the system reverts to the traditional network stack with the associated overheads for larger connection rates. For busy network servers with a large number of short-lived connections, such offload does not offer significant benefit.

Muir et. al. propose Piglet [11], where the device driver functionality is offloaded to the dedicated processors. Embedded Transport Acceleration (ETA) dedicates one or more processors or hardware threads to perform all network processing [15, 3]. Unlike the above, where the set of dedicated processors is fixed and static, TCPServers monitors the system load and uses a reconfigurable dedicated processor set. It also uses both interrupts and polling to handle NIC events. Finally, TCPServers implements a scheduling discipline that handles network processing at the priority of the destination socket

7. Conclusions

In this paper, we presented techniques to improve concurrency of network processing using TCPServers architecture. We used early demultiplexing of incoming network packets in conjunction with per-socket fixed sized Receive Queues, in order to eliminate access to globally shared connection data structures in the critical path. We also present a proportional fair scheduling algorithm that processes network packets according to the priority of their destination socket. Our evaluation shows that with our modifications, TCPServers improves the aggregate TCP/IP throughput by up to 75% compared to the default SMP implementation.

References

[1] Apache HTTP Server. http://httpd.apache.org.

[2] M. Bjorkman and P. Gunningberg. Performance Modeling of Multiprocessor Implementations of Protocols. *IEEE/ACM Trans. Netw.*, 6(3):262–273, 1998.

[3] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O. In *Proc. of EuroSys 2006*, pages 265–278, Leuven, Belgium, April 2006.

[4] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. In *Proc. of Usenix Annual Tech. Conference*, Anaheim, CA, April 2005.

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

[6] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. M. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In *Proc. of Usenix Annual Tech. Conference*, Anaheim, CA, April 2005.

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

[8] H. Kim and S. Rixner. TCP Offload through Connection Handoff. In *Proc. of Eurosys 2006*, Leuven, Belgium, April 2006.

[9] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, pages 99–111, Berkeley, CA, USA, Jan. 1996.

[10] D. Mosberger and T. Jin. httperf – a tool for measureing web server performance. In *Proc. SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, 1998.

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

[12] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance Isses in Parallelized Network Protocols. In *Proc. of Symposium on Operating Systems Design and Implementation*, Nov 1994.

[13] N. Provos. libevent - An Event Notification Library. http://monkey.org/ provos/libevent/.

[14] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, L. Iftode, and W. Zwaenepoel. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance. Technical Report TR-481, Rutgers University, March 2001.

[15] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahulli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *IEEE Computer*, 37(11):48–58, 2004.

[16] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proc. 5th Annual Linux Showcase and Conference*, pages 165–172, Oakland, CA, Nov 2001.

[17] D. C. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *Proc. of IEEE INFOCOM*, pages 624–633, 1995.

[18] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proc. Usenix Annual Technical Conference*, Boston, MA, June 2006.

[19] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. In *Proc. of ACM SIGMETRICS 1996*, pages 116–125, May 1996.