An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems

Paul Willmann, Scott Rixner, and Alan L. Cox
Rice University
{willmann, rixner, alc}@rice.edu

Abstract

As technology trends push future microprocessors toward chip multiprocessor designs, operating system network stacks must be parallelized in order to keep pace with improvements in network bandwidth. There are two competing strategies for stack parallelization. Message-parallel network stacks treat messages (usually packets) as the fundamental unit of concurrency, whereas connection-parallel network stacks treat connections as the fundamental unit of concurrency. Practical implementations of connection-parallel stacks map operations to groups of connections and permit concurrent processing on independent connection groups, thus treating the group as the unit of concurrency. Connection-parallel stacks can use either locks or threads to serialize access to connection groups. This paper evaluates these parallel stack organizations using a modern operating system and chip multiprocessor hardware.

Compared to uniprocessor kernels, all parallel stack organizations incur additional locking overhead, cache inefficiencies, and scheduling overhead. However, the organizations balance these limitations differently, leading to variations in peak performance and connection scalability. Lock-serialized connection-parallel organizations reduce the locking overhead of message-parallel organizations by using many connection groups and eliminate the expensive thread handoff mechanism of thread-serialized connection-parallel organizations. The resultant organization outperforms the others, delivering 5.4 Gb/s of TCP throughput for most connection loads and providing a 126% throughput improvement versus a uniprocessor for the heaviest connection loads tested, utilizing 16384 simultaneous connections.

1 Introduction

Network bandwidths continue to increase at an exponential pace, and the demand for network content shows no sign of letting up. In the past, exponential gains in microprocessor performance have always allowed processing power to catch up with network bandwidth. However, the complexity of modern uniprocessors will prevent such continued performance growth. Instead, microprocessors have begun to provide parallel processing cores in order to make up for the loss in performance growth of individual processor cores. In order for network servers to exploit these parallel processors, scalable parallelizations of the network stack are needed.

Modern network stacks can exploit either message-based parallelism or connection-based parallelism. Network stacks that exploit message-based parallelism, such as Linux and FreeBSD, allow multiple threads to simultaneously process different messages from the same or different connections. Network stacks that exploit connection-based parallelism, such as DragonflyBSD and Solaris 10 [18], assign each connection to a group. Threads may then simultaneously process messages as long as they belong to different connection groups. The connection-based approach can use either threads or locks for synchronization, yielding three major parallel network stack organizations: message-based (MsgP), connection-based using threads for synchronization (ConnP-T), and connection-based using locks for synchronization (ConnP-L). These competing parallelization strategies are implemented within the FreeBSD 7 operating system to enable a fair comparison.

The uniprocessor version of FreeBSD is efficient, but its performance falls short of saturating available network resources in a modern machine and degrades significantly as connections are added. Utilizing 4 cores, the parallel stack organizations can outperform the uniprocessor stack (especially at high connection loads), but

A shorter version of this work appeared at the USENIX 2006 Annual Technical Conference

each parallel stack organization incurs higher locking overhead, reduced cache efficiency, and higher scheduling overhead than the uniprocessor. MsgP outperforms the uniprocessor for almost all connection loads but experiences significant locking overhead. In contrast, ConnP-T has very low locking overhead but incurs significant scheduling overhead, leading to reduced performance compared to even the uniprocessor kernel for all but the heaviest loads. ConnP-L mitigates the locking overhead of MsgP, by grouping connections so that there is little global locking, and the scheduling overhead of ConnP-T, by using the requesting thread for network processing rather than forwarding the request to another thread. This results in the best performance of all stacks considered, delivering stable performance of 5440 Mb/s for moderate connection loads and providing a 126% improvement over the uniprocessor kernel when utilizing 16384 simultaneous connections.

The remainder of this paper proceeds as follows. The next section further motivates the need for parallelized network stacks in current and future systems and discusses previous evaluations of parallel network stack organizations. Section 3 describes the parallel network stack architectures that are evaluated in this paper. Section 4 then presents an evaluation of each organization and Section 5 discusses these results. Finally, Section 6 concludes the paper.

2 Background

The most efficient network stacks in modern operating systems are single-threaded, forcing the network stack to only run on a single processor core at a time. If two threads need to perform network processing simultaneously, synchronization is necessary. However, independent threads can run concurrently on separate processors, as long as they do not access the operating system simultaneously. For multithreaded server applications that require a large amount of user-level processing, this level of parallelism may be sufficient, as the operating system's network processing capabilities are unlikely to be a performance bottleneck.

2.1 Performance Scaling

For network-intensive applications, however, such single-threaded network stacks are not capable of saturating a modern 10 Gbps Ethernet link. Hurwitz and Feng found that, using Linux 2.4 and 2.5 uniprocessor kernels (with TCP segmentation offloading), they were only able to achieve about 2.5 Gbps on a 2.4 GHz Intel Pentium 4 Xeon system [4]. Hence, network stack processing currently represents a bottleneck for leveraging the physically achievable network throughput on modern

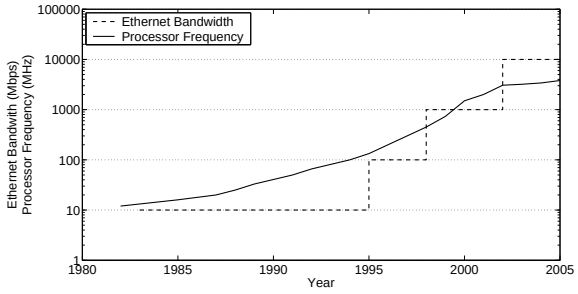

Figure 1: Intel microprocessor frequency trends in relation to Ethernet bandwidth growth.

links. Though the use of jumbo frames can improve these numbers, the need for interoperability with legacy hardware dictates that most network servers will use standard 1500 byte Ethernet frames.

While it has traditionally been the case that uniprocessors have not been able to saturate the network with the introduction of each new bandwidth generation, exponential gains in uniprocessor performance have always allowed processing power to catch up with network bandwidth. However, it is very likely that this will no longer be the case in the future.

Figure 1 shows the growth in Ethernet bandwidth and processor frequency over time. The graph plots Ethernet bandwidth in Mbps based upon when new Ethernet standards are ratified, and the highest processor frequency of Intel chips in MHz. Processor frequency is obviously not equivalent to performance, but Intel processor frequency has been a reasonable approximation of processor performance trends over the period in the graph. As the graph shows, both Ethernet bandwidth and processor frequency have been growing exponentially. However, processor frequency growth has not kept up with Ethernet bandwidth growth in the past 5 years. Furthermore, processor frequencies have largely leveled out recently. Clearly, processor performance continues to improve, but the rate of improvement has definitely slowed in recent years.

The complexity of modern uniprocessors has made it prohibitively expensive to continue to improve processor performance at the same rate as in the past. Not only is it difficult to further increase clock frequencies, as Figure 1 implies, but it is also difficult to further improve the efficiency of modern uniprocessor architectures [1]. Therefore, industry has turned to single chip multiprocessors (CMPs) [14] to continue to increase processor performance. One of the first dual-core chips was the IBM Power 4 architecture [3, 17]. The IBM Power 4 includes two complete processing cores, allowing roughly twice the performance of a single core for two independent threads of control. Shortly after the release of the Power 4, Intel introduced hyperthreading in the Pentium

4, enabling two threads to utilize a single core more efficiently [9]. While hyperthreading does not duplicate the processor core, it does provide modest speedups for two independent threads, as the second thread can take advantage of resources unused by the first thread. Subsequently, Sun, AMD, and Intel have all released dual-core processors [5, 10, 11].

This indicates a clear trend towards single-chip multiprocessing. Sun's Niagara is perhaps the most aggressive example, with 8 cores on a single chip, each capable of executing four threads of control [8, 12]. Unfortunately, each processor in a single-chip multiprocessor is unlikely to perform as well as a monolithic uniprocessor. For example, the fastest dual-core Opteron processors operate at lower frequencies than their single-core counterparts. This is likely to result in an even greater performance loss due to shared resources between the cores. It seems clear that networking code will have to be parallelized extensively in order to saturate the network with these parallel processing cores.

2.2 Network Stack Parallelization

Since uniprocessor performance will not improve fast enough to saturate future networks, and most server applications manage multiple network connections concurrently using separate threads, operating systems have begun to exploit connection-level parallelism by allowing multiple threads to carry out network operations concurrently in the kernel. However, supporting this parallelism comes with significant cost [2, 4, 13, 15, 20]. For example, uniprocessor Linux kernels deliver 20% better end-to-end throughput over 10 Gigabit Ethernet than multiprocessor kernels [4]. Since the performance of individual processors will not increase nearly as fast as they have in the past as microprocessors move towards CMP architectures, there is a significant need to improve the efficiency of multiprocessor kernels for networking applications.

Much of the research on parallelizing the network stack took place in the mid-1990s, between the introduction of 100 Mbps Ethernet and Gigabit Ethernet. At that time, two forms of network processing parallelism were examined: message-oriented and connection-oriented parallelism. Using message-oriented parallelism, any message (or packet) may be processed simultaneously on a separate thread. Hence, messages for a single connection could be processed concurrently on different threads, potentially resulting in improved performance. Connection-oriented parallelism is more coarse-grained; at the beginning of network processing (either at the top or bottom of the network stack), messages and packets are classified according to the connection with which they are associated. All packets for a certain connection are then processed by a specific thread; a thread may be responsible for processing one or more specific connections.

Nahum et al. first examined message-oriented parallelism on an SGI Challenge shared-memory multiprocessor [13]. That study examined the behavior of a message-parallel implementation of the user-space *x*-kernel utilizing a memory-only pseudo network device. They studied the effects of locking granularity within the network stack and overall stack scalability with respect to the number of processors; because message-parallel stacks allow messages from the same connection to be processed simultaneously, mutual exclusion is required to ensure that higher-level connection state (such as reassembly queues and timeout state) is kept consistent. This study found that, in general, implementing finer grained locking around connection state variables degrades performance by introducing additional overhead and does not result in significant improvements in speedup. In that case, coarser-grained locking (with just one lock protecting all TCP state) performed best. Furthermore, they found that careful attention had to be paid to thread scheduling and lock acquisition ordering on the receive side to ensure that packets that arrived in order did in fact get processed in order.

Yates et al. later examined a connection-oriented parallel implementation of the *x*-kernel, also utilizing a pseudo network device and running on the SGI Challenge architecture [20]. That study examined the effects of scaling the number of threads with the number of connections; they found that increasing the number of threads to match the number of connections yielded the best results, even when the number of threads was far greater than the number of physical processors. At the time, the OS limited the user-space *x*-kernel to 384 threads; Yates et al. propose using as many as are supported by the system.

Schmidt and Suda compared message-oriented and connection-oriented network stacks in a modified version of SunOS utilizing a real network interface [16]. They found that, given just a few connections, a connection-parallel stack outperforms a message-parallel one. However, they note that the number of context switches increases significantly as connections (and processors) are added to the connection-parallel scheme, and that the cost of synchronization heavily affects the efficiency with which each scheme operates (especially the message-parallel scheme).

The costs of synchronization and context switches have changed dramatically in the years since many of the seminal works that examined network stack parallelization strategies. Though processors have become much faster, the gap between memory system performance and processing performance has become much greater, mak-

ing the worst-case cost of synchronization much higher in terms of lost execution cycles. Moreover, this gap exacerbates the cost of a context switch as thread state is swapped in memory. Both the need to close the gap between computation and physically achievable Ethernet throughput, and the vast changes in the architectural characteristics that shaped prior analyses of parallel network stacks motivate a fresh examination of parallel network stack architectures on modern parallel hardware.

3 Parallel Network Stack Architectures

Despite the conclusions of the 1990s described in the previous section, there still does not seem to be a solid consensus among modern operating system developers on how to design efficient and scalable parallel network stacks. Major subsystems of FreeBSD and Linux, including the network stack, have been redesigned in recent years to improve performance on parallel hardware. Both operating systems now incorporate variations of message-based parallelism within their network stacks. Conversely, Sun has recently rearchitected the Solaris operating system in anticipation of their high-throughput computing microprocessors and now incorporates a variation of connection-based parallelism [18], as does DragonflyBSD.

These operating system improvements are directly targeted at the architectural trend towards increasingly parallel hardware, including multiple processing cores and threads of control on a single chip, as described in Section 2.1. While the different design choices may be motivated by a variety of factors, it is interesting that operating systems developers have not unanimously chosen to follow the recommendations of past research. Therefore, a reevaluation of network stack parallelization techniques in the context of modern operating systems and hardware is warranted to uncover the benefits and pitfalls of the different organizations.

FreeBSD is chosen as a representative modern operating system for the purposes of this evaluation. FreeBSD implements a competitively parallelized network stack based upon a variation of message-based parallelism (MsgP), and it includes kernel-adaptive mutexes and intra-kernel threading, which are useful building blocks for an efficient implementation of a connection-based parallel (ConnP) network stack. Furthermore, the other major subsystems are also parallelized competitively and thus should not introduce biasing bottlenecks. Therefore, the FreeBSD 7 operating system was modified to include a connection-based parallel network stack to compare to its message-based parallel network stack. The following subsections describe the architecture of both network stacks.

3.1 Message-based Parallelism (MsgP)

Message-based parallel (MsgP) network stacks, such as FreeBSD, exploit parallelism by allowing multiple threads to operate within the network stack simultaneously. Two types of threads may perform network processing: one or more application threads and one or more inbound protocol threads. When an application thread makes a system call, that calling thread context is "borrowed" to then enter the kernel and carry out the requested service. So, for example, a `read` or `write` call on a socket would loan the application thread to the operating system to perform networking tasks. Multiple such application threads can be executing within the kernel at any given time. The network interface's driver executes on an inbound protocol thread whenever the network interface card (NIC) interrupts the host, and it may transfer packets between the NIC and host memory. After servicing the NIC, the inbound protocol thread processes received packets "up" through the network stack.

Given that multiple threads can be active within the network stack, FreeBSD utilizes fine-grained locking around shared kernel structures to ensure proper message ordering and connection state consistency. As a thread attempts to send or receive a message on a connection, it must acquire various kernel-adaptive locks when accessing shared connection state, such as the global connection hash table lock (for looking up TCP connections) and per-connection locks (for both socket state and TCP state). If a thread is unable to obtain a lock, it is placed in the lock's queue of waiting threads and yields the processor, allowing another thread to execute. To prevent priority inversion, priority propagation from the waiting threads to the thread holding the lock is performed.

As is characteristic of message-based parallel network stacks, FreeBSD's locking organization thus allows concurrent processing of different messages on the same connection, so long as the various threads are not accessing the same portion of the connection state at the same time. For example, one thread may process TCP timeout state based on the reception of a new ACK, while at the same time another thread is copying data into that connection's socket buffer for later transmission. However, note that the inbound thread configuration described is not the FreeBSD 7 default. Rather, the operating system's network stack has been configured to use the optional `direct_dispatch` mechanism. Normally dedicated parallel driver threads service each NIC and then hand off inbound packets to a single worker thread via a shared queue. That worker thread then processes the received packets "up" through the network stack. The default configuration thus limits the performance of MsgP and is hence not considered in this paper. The thread-per-NIC model also differs from the message-parallel or-

ganization described by Nahum *et al.* [13], which used many more worker threads than interfaces. Such an organization requires a sophisticated scheme to ensure these worker threads do not reorder inbound packets that were received in order, and hence that organization is also not considered.

3.2 Connection-based Parallelism (ConnP)

To compare connection parallelism in the same framework as message parallelism, FreeBSD 7 was modified to support two variants of connection-based parallelism (ConnP) that differ in how they serialize TCP/IP processing within a connection. The first variant assigns each connection to one of a small number of protocol processing threads (ConnP-T). The second variant assigns each connection to one of a small number of locks (ConnP-L).

3.2.1 Connection Parallelism Serialized by Threads (ConnP-T)

Connection-based parallelism using threads utilizes several kernel threads dedicated to per-connection protocol processing. Each protocol thread is responsible for processing a subset of the system's connections. At each entry point into the TCP/IP protocol stack, the requested operation is enqueued for service by a particular protocol thread based on the connection that is being processed. Each connection is uniquely mapped to a single protocol thread for the lifetime of that connection. Later, the protocol threads dequeue requests and process them appropriately. No per-connection state locking is required within the TCP/IP protocol stack, because the state of each connection is only manipulated by a single protocol thread.

The kernel protocol threads are simply worker threads that are bound to a specific CPU. They dequeue requests and perform the appropriate processing; the messaging system between the threads requesting service and kernel protocol threads maintains strict FIFO ordering. Within each protocol thread, several data structures that are normally system-wide (such as the TCP connection hash table) are replicated so that they are thread-private. Kernel protocol threads provide both synchronous and asynchronous interfaces to threads requesting service.

If a requesting thread requires a return value or if the requester must maintain synchronous semantics (that is, the requester must wait until the kernel thread completes the desired request), that requester uses a condition variable to yield the processor and wait for the kernel thread to complete the requested work. Once the kernel protocol thread completes the desired function, the kernel thread fills in the return value into the message that was passed to it and then signals the waiting thread. This is the common case for application threads, which require a return value to determine if the network request succeeded. However, interrupt threads (such as those that service the network interface card and pass "up" packets received on the network) do not require synchronous semantics. In this case, the interrupt context classifies each packet according to its connection and enqueues the packet for the appropriate kernel protocol thread.

The connection-based parallel stack uniquely maps a packet or socket request to a specific protocol thread by hashing the 4-tuple of remote IP address, remote port number, local IP address, and local port number. However, not every portion of this tuple is defined for every protocol operation. For example, the `listen()` call does not have a remote address or port number associated with it. Such operations that cannot be uniquely associated with a specific connection are processed on protocol thread 0. In the case of `listen()`, a wildcard entry must be made in each protocol thread's connection hash table, because any remote IP address and port number combination could arrive at the local machine, resulting in a hash to any of the local kernel protocol threads. When that happens, any of the protocol threads must be able to find the wildcard entry in its local hash table. Similarly, `close()` may require a shoot-down of all wildcard entries on remote threads. Hence, though locking overheads are reduced within the protocol stack in the common case, some start-up and tear-down operations are more expensive because they require messaging that scales with the number of kernel protocol threads. This implementation of connection-oriented parallelism is like that of DragonflyBSD.

3.2.2 Connection Parallelism Serialized by Locks (ConnP-L)

Just as in thread-serialized connection parallelism, connection-based parallelism using locks is based upon the principle of isolating connections into groups that are each bound to a single entity during execution. As the name implies, however, the binding entity is not a thread; instead, each group is isolated by a mutual exclusion lock.

When an application thread enters the kernel to obtain service from the network stack, the network system call maps the connection being serviced to a particular group using a mechanism identical to that employed by thread-serialized connection parallelism. However, rather than building a message and passing it to that group's specific kernel protocol thread for service, the calling thread directly obtains the lock for the group associated with the given connection. After that point, the calling thread may access any of the group-private data structures, such as the group-private connection hash table or group-private

per-connection structures. Hence, these locks serve to ensure that at most one thread may be accessing each group's private connection structures at a time. Upon completion of the system call in the network stack, the calling thread releases the group lock, allowing another thread to obtain that group's lock if necessary. Threads accessing connections in different groups may proceed concurrently through the network stack without obtaining any stack-specific locks other than the group lock.

Inbound packet processing is also analogous to connection-based parallelism using threads. After receiving a packet, the inbound protocol thread classifies the packet into a group. Unlike the thread-oriented connection-parallel case, the inbound thread need not hand off the packet from the driver to the worker thread corresponding to the packet's connection group. Instead, the inbound thread directly obtains the appropriate group lock for the packet and then processes the packet "up" the protocol stack without any thread handoff. This control flow is similar to the message-parallel stack, but the lock-serialized connection-parallel stack does not require any further protocol locks after obtaining the connection group lock. As in the MsgP case, there is one inbound protocol thread for each NIC, but the number of groups may far exceed the number of threads.

This implementation of connection-oriented parallelism is similar to Solaris 10. At the core of Solaris 10's implementation is the `squeue` abstraction, which is short for "serialization queue". The `squeue` consists of a packet queue, a worker thread, a set of flags, and a lock. The worker thread is bound to a processor core (or in the case of a multithreaded processor microarchitecture, a virtual processor). Each connection is permanently associated with an `squeue` at its creation. The number of `squeue`s is equal to the number of processor cores (or virtual processors). Once the connection to which an inbound or outbound packet belongs is determined, the packet is queued on that connection's `squeue` for TCP/IP processing. The thread that enqueued the packet then either (1) does nothing, (2) wakes up the worker thread to perform TCP/IP processing on the packet queue, or (3) attempts to acquire exclusive control of the `squeue` and itself perform TCP/IP processing on the packet queue. In any case, at most one thread is ever performing TCP/IP processing on the packet queue at a time. The implementations evaluated in this work are slightly more rigidly defined. For lock-serialized connection parallelism (ConnP-L), application and inbound protocol threads always acquire exclusive control of the group lock and carry out stack processing directly. For thread-serialized connection parallelism (ConnP-T), application and inbound driver threads always hand off network requests and inbound packets to a worker thread for further processing.

4 Evaluation

To gain insights into the behavior and characteristics of the parallel network stack architectures described in Section 3, these architectures were evaluated on a modern chip multiprocessor. All stack architectures were implemented within the 2006-03-27 repository version of the FreeBSD 7 operating system to facilitate a fair comparison. This section describes the benchmarking methodology and hardware platforms and presents the scalability of the network stack organizations.

4.1 Evaluation Hardware

The parallel network stack organizations were evaluated using a 4-way SMP AMD Opteron system. The system consists of a Tyan S2885 motherboard, two dual-core Opteron 275 processors, two 1 GB PC2700 DIMMs per processor (one per memory channel), and three dual-port Intel PRO/1000-MT Gigabit Ethernet network interfaces spread across the motherboard's PCI-X bus segments. Data is transferred between the 4-way Opteron's 6 Ethernet interfaces and three client systems. Each client has two Gigabit Ethernet interfaces and uses faster processors and memory. Each of these clients is directly connected to the 4-way Opteron without use of a switch, and each can independently sustain the theoretical peak bandwidth of its two interfaces. Therefore, all results are determined solely by the behavior of the 4-way Opteron 275 system. The Opterons' hardware performance counters are used to perform low-overhead profiling.

4.2 Parallel TCP Benchmark

Most existing network benchmarks evaluate single-connection performance. However, modern multithreaded server applications simultaneously manage tens to thousands of connections. This parallel network traffic behaves quite differently than a single network connection. To address this issue, a multithreaded, event-driven, network benchmark was developed that distributes traffic across a configurable number of connections. The benchmark distributes connections evenly across threads and utilizes `libevent` to manage connections within a thread. For all of the experiments in this paper, the number of threads used by the benchmark is equal to the number of processor cores. Each thread manages an equal number of connections. For experiments using more than 24 connections, the application's connections are distributed across the server's 6 NICs equally such that each of the four threads uses each NIC, and every thread has the same number of connections that map to each NIC.

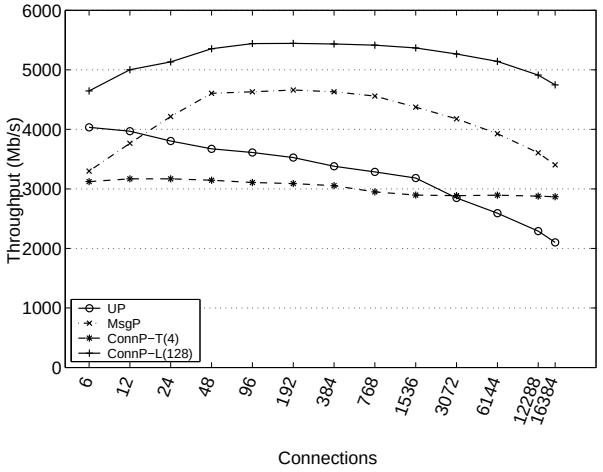

Figure 2: Aggregate throughput for message-parallel and connection-parallel network stacks.

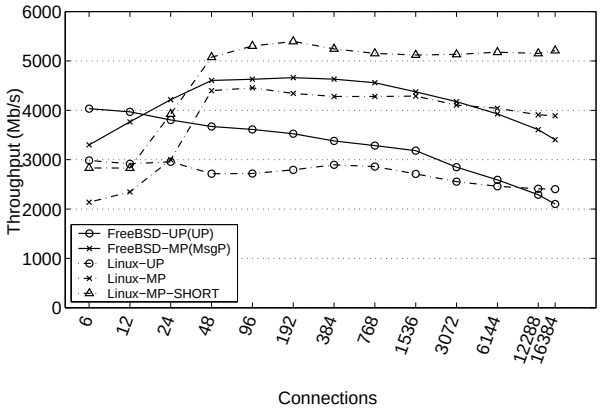

Figure 3: Aggregate throughput comparison for FreeBSD and Linux.

Each thread sends data over all of its connections using a per-thread file using zero-copy `sendfile()`. The sending and receiving socket buffer sizes are set to be 256 KB in all tests to accomodate the large TCP windows on high-bandwidth links. Using larger socket buffers did not improve performance for any test. Furthermore, all experiments use the standard 1500-byte maximum transmission unit and do not utilize TCP segmentation offload, which currently is not implemented in FreeBSD. The benchmark is always run for 3 minutes unless otherwise noted.

4.3 Network Throughput

Figure 2 depicts the aggregate throughput across all connections when executing the parallel TCP benchmark described in Section 4.2 utilizing various configurations of FreeBSD 7. "UP" is the uniprocessor version of the FreeBSD kernel running on a single core of the Opteron server. The rest of the configurations are run on all 4 cores. "MsgP" is the multiprocessor FreeBSD-based MsgP kernel described in Section 3.1. MsgP uses a lock per connection. "ConnP-T(4)" is the multiprocessor FreeBSD-based ConnP-T kernel described in Section 3.2.1, using 4 kernel protocol threads for TCP/IP stack processing that are each pinned to a different core. "ConnP-L(128)" is the multiprocessor FreeBSD-based ConnP-L kernel described in Section 3.2.2. ConnP-L(128) divides the connections among 128 locks within the TCP/IP stack.

The figure shows that the "UP" kernel performs well with a small number of connections, achieving a bandwidth of 4034 Mb/s with only 6 connections. However, total bandwidth decreases as the number of connections increases. MsgP achieves 82% of the uniprocessor bandwidth at 6 connections but quickly ramps up to 4630 Mb/s, holding steady through 768 connections and then decreasing to 3403 Mb/s with 16384 connections. ConnP-T(4) achieves close to its peak bandwidth of 3123 Mb/s with 6 connections and provides approximately steady bandwidth as the number of connections increase. Finally, the ConnP-L(128) curve is shaped similar to that of MsgP, but its performance is larger in magnitude and always outperforms the uniprocessor kernel. ConnP-L(128) delivers steady performance around 5440 Mb/s for 96–768 connections and then gradually decreases to 4747 Mb/s with 16384 connections. This peak performance is roughly the peak TCP throughput deliverable by the three dual-port Gigabit NICs.

To demonstrate that FreeBSD is representative in performance of modern message-parallel operating systems, Figure 3 compares the throughput of the FreeBSD-based uniprocessor ("UP") and multiprocessor ("MsgP") operating systems against the corresponding Linux-based variants. The Linux configurations used kernel version 2.6.16.15, ran on the same hardware, and used the same TCP stack tuning parameters. The `sendfile()`-based microbenchmark application was modified to use the Linux `TCP_CORK` socket option to optimize performance. TCP segmentation offloading was disabled for these tests because it led to connection resets under loads greater than about 1000 simultaneous connections and prevented tests from completing successfully. In Figure 3, "FreeBSD-UP(UP)" and "FreeBSD-MP(MsgP)" are the uniprocessor and multiprocessor versions of the FreeBSD 7 operating system, also depicted as "UP" and "MsgP" in Figure 2. Figure 3 also depicts the uniprocessor version of Linux ("Linux-UP"), the multiprocessor version of Linux using 4 cores ("Linux-MP"), and the multiprocessor version of Linux using 4 cores and running a shorter, 20-second test ("Linux-MP-SHORT").

Figure 3 shows that for the standard test duration,

FreeBSD and Linux perform comparably in both uniprocessor and multiprocessor configurations for most connection loads tested, though uniprocessor FreeBSD outperforms uniprocessor Linux by up to 1 Gb/s for smaller connection loads. Hence, FreeBSD is a competitive platform to use as a basis for implementing other network stack architectures. The 20-second multiprocessor Linux test ("Linux-MP-SHORT") showed a significant improvement over the 3-minute multiprocessor Linux and FreeBSD tests, however. Clearly the test duration has a significant impact on aggregate throughput. This relationship was not observed using uniprocessor Linux or any configuration of FreeBSD, and it remains unclear what is causing this anomaly in the multiprocessor Linux case.

5 Discussion and Analysis

Figure 2 shows that using 4 cores, ConnP-L(128) and MsgP outperform the uniprocessor FreeBSD 7 kernel for almost all connection loads. As mentioned in Section 2, the use of TCP segmentation offloading and/or jumbo frames can further improve the peak performance, but these optimizations should benefit each organization approximately equally. Regardless, the speedup attained using multiprocessor kernels is significantly less than ideal and is limited by (1) locking overhead, (2) cache efficiency, and (3) scheduling overhead. The following subsections will explain how these issues affect the parallel implementations of the network stack.

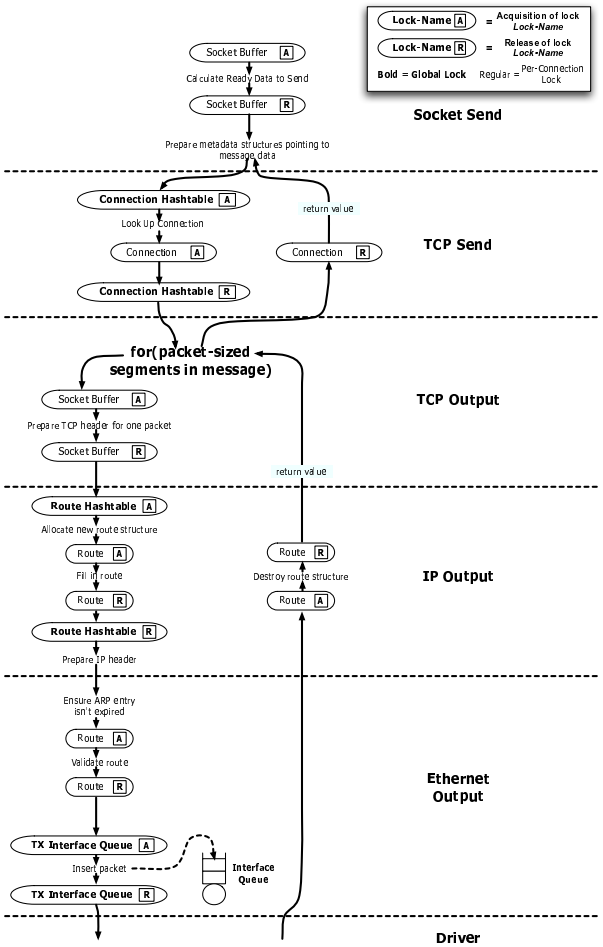

Figure 4: The Outbound Control Path in the Application Thread Context.

5.1 Locking Overhead

There are two significant costs of locking within the parallelized network stacks. The first is that SMP locks are fundamentally more expensive than uniprocessor locks. In a uniprocessor kernel, a simple atomic test-and-set instruction can be used to protect against interference across context switches, whereas, SMP systems must use system wide locking to ensure proper synchronization among simultaneously running threads. This is likely to incur significant overhead in the SMP case. For example, on x86 architectures, the `lock` prefix, which is used to ensure that an instruction is executed atomically across the system, effectively locks all other cores out of the memory system during the execution of the locked instruction.

The second is that contention for global locks within the network stacks is significantly increased when multiple threads are actively performing network tasks simultaneously. Figure 4 shows the locking required in the control path for send processing within the sending application's thread context in the MsgP network stack of FreeBSD 7. Most of the locks pictured are associated with a single socket buffer or connection. Therefore, it is unlikely that multiple application threads would contend for those locks since it does not make sense to use multiple application threads to send data over the same connection. However, those locks could be shared with the kernel's inbound protocol threads that are processing receive traffic on the same connection. Global locks that must be acquired by all threads that are sending (or possibly receiving) data over any connection are far more problematic.

There are two global locks on the send path: the `Connection Hashtable` lock and the `Route Hashtable` lock. These locks protect the hash table that maps a particular connection to its individual connection lock and the hash table that maps a particular connection to its individual route lock, respectively. These locks are also used in lieu of explicit reference counting for individual connections and locks. Watson presents a more detailed description of locking within the FreeBSD network stack [19].

OS Type	6 conns	192 conns	16384 conns
MsgP	89	100	100
ConnP-L(4)	60	56	52
ConnP-L(8)	51	30	26
ConnP-L(16)	49	18	14
ConnP-L(32)	41	10	7
ConnP-L(64)	37	6	4
ConnP-L(128)	33	5	2

Table 1: Percentage of lock acquisitions for global TCP/IP locks (`Connection Hashtable` and `Network Group` locks) that do not succeed immediately.

There is very little contention for the `Route Hashtable` lock because the corresponding `Route` lock is quickly acquired and released so a thread is unlikely to be blocked while holding the `Route Hashtable` lock and waiting for a `Route` lock. In contrast, the `Connection Hashtable` lock is highly contended. This lock must be acquired by any thread performing any network operation on any connection. Furthermore, it is possible for a thread to block while holding the lock and waiting for its corresponding `Connection` lock, which can be held for quite some time.

Table 1 depicts global TCP/IP lock contention, measured as the percentage of lock acquisitions that do not immediately succeed because another thread holds the lock. ConnP-T is omitted from the table because it eliminates global TCP/IP locking completely. As the table shows, the MsgP network stack experiences significant contention for the `Connection Hashtable` lock, which leads to considerable overhead as the number of connections increases.

One would expect that as connections are added, the probability of contention for per-connection locks would decrease, and in fact lock profiling supports this conclusion. However, because other locks (such as that guarding the scheduler) are acquired while holding the per-connection lock, and because those other locks are system-wide and become highly contended during heavy loads, detailed locking profiles show that the average amount of time that each per-connection lock is held increases dramatically. Hence, though contention for per-connection locks decreases, the increasing cost for a contended lock is so much greater that the system exhibits increasing average acquisition times for per-connection locks as connections are added. This increased per-connection acquisition time in turn leads to longer waits for the `Connection Hashtable` lock, eventually bogging down the system with contention.

Whereas the MsgP stack relies on repeated acquisition to the `Connection Hashtable` and `Connection` locks to continue stack processing, ConnP-L stacks

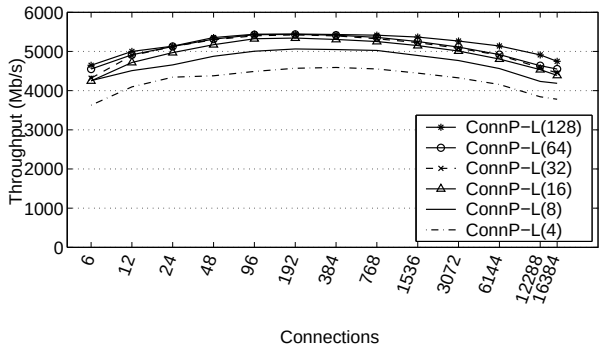

Figure 5: Aggregate throughput for the connection-based, parallel network stack using locks as the number of locks is varied.

can also become periodically bottlenecked if a single group becomes highly contended. As discussed in Section 3.2.2, the number of connection groups in a ConnP-L network stack may exceed the number of inbound protocol threads. Table 1 shows the contention for the `Network Group` locks for ConnP-L stacks as the number of network groups is varied to from 4 to 128 groups. The table demonstrates that contention for the `Network Group` locks consistently decreases as the number of network groups increases. Though ConnP-L(4)'s `Network Group` lock contention is high at over 50% for all connection loads, increasing the number of network groups to 128 reduces contention from 52% to just 2% for the heaviest connection load. Note that contention decreases for the 6-connection case even as the number of `Network Group` locks increases well beyond 6. The signficant contention decreases of more than one or two percent are caused by better hashing distribution of connections to connection groups. That is, as more groups are added, the static hashing function that assigns connections to connection groups is more likely to assign each connection to a separate group. The more modest contention decreases are caused by reductions in contention by book-keeping functions that update all connection groups and thus access all `Network Group` locks. For example, updates to TCP syncache entries are lazily broadcast to all groups via a timer-based mechanism, regardless of whether or not they currently have an active connection. These updates are generally not contended, but because there are more of them as the number of groups increases, `Network Group` contention decreases.

Figure 5 shows the effect increasing the number of network groups has on aggregate throughput for 6, 192, and 16384 connections. As is suggested by the contention reduction associated with larger numbers of network groups, network throughput increases with more

OS Type	6 conns	192 conn	16384 conns
UP	1.8	4.1	18.5
MsgP	37.3	28.4	40.5
ConnP-T(4)	52.3	50.4	51.4
ConnP-L(128)	28.9	26.2	40.4

Table 2: **L2 Data cache misses per 1 Kilobyte of payload data transmitted.**

OS Type	6 conns	192 conns	16384 conns
UP	482	440	423
MsgP	2904	1818	2448
ConnP-T(4)	3488	3602	4535
ConnP-L(128)	2135	924	1064

Table 3: Number of cycles spent managing the scheduler and scheduler synchronization per Kilobyte of data transmitted.

network groups. However, there are diminishing returns as more groups are added.

5.2 Cache Behavior

Table 2 shows the number of L2 data cache misses per KB of payload data transmitted collected using the Opterons' performance counters. Figure 6 plots these L2 misses categorized according to where in the operating system the misses occurred (e.g., in the network stack or in the thread scheduler). This data shows the efficiency of the cache hierarchy normalized to network bandwidth. The uniprocessor kernel incurs very few cache misses relative to the multiprocessor configurations. The lack of metadata migration accounts for the uniprocessor kernel's cache efficiency. The increase in the amount of connection state within the kernel stresses the cache and directly results in increased cache misses and decreased throughput as the number of connections are increased [6, 7].

The parallel network stacks incur significantly more cache misses per KB of transmitted data because of data migration and lock accesses. Surprisingly, ConnP-T(4) incurs the most cache misses despite each thread being pinned to a specific processor core. One might expect that such pinning would improve locality by eliminating migration of many connection data structures. However, Figure 6 shows that for the cases with 6 and 192 connections, ConnP-T(4) exhibits more misses in the network stack than any of the other organizations. While thread pinning can improve locality by eliminating migration of connection metadata, frequently updated socket metadata is still shared between the application and protocol threads, which leads to data migration and a higher cache miss rate. Pinning the protocol threads does result in better utilization of the caches for the 16384-connection load, however. In this case, ConnP-T(4) exhibits the fewest network-stack L2 cache misses. However, the relatively higher number of L2 cache misses caused by the scheduler prevents this advantage from translating into a performance benefit.

5.3 Scheduler Overhead

The ConnP-T kernel trades the locking overhead of the ConnP-L and MsgP kernels for scheduling overhead. As operations are requested for a particular connection, they must be scheduled onto the appropriate protocol thread. As Figure 2 showed, this results in stable, but low total bandwidth as connections scale for ConnP-T(4). ConnP-L approximates the reduced intra-stack locking properties of ConnP-T and adopts the simpler scheduling properties of MsgP; locking overhead is minimized by the additional groups and scheduling overhead is minimized since messages are not transferred to protocol threads. This results in consistently better performance than the other parallel organizations.

To further explain this behavior, Table 3 shows the number of cycles spent managing the scheduler and scheduler synchronization per KB of payload data transmitted collected using the Opterons' performance counters. This shows the overhead of the scheduler normalized to network bandwidth. Though MsgP experiences less scheduling overhead as the number of connections increase and threads aggregate more work, locking overhead within the threads quickly negate the scheduler advantage. In contrast, the scheduler overhead of ConnP-T remains high, corresponding to relatively low bandwidth. In contrast, ConnP-L exhibits stable scheduler overhead that is much lower than ConnP-T and MsgP, contributing to its higher throughput. ConnP-L does not require a thread handoff mechanism and its low lock contention compared to MsgP results in fewer context switches from threads waiting for locks.

Both Table 3 and Figure 6 show that the reference ConnP-T implementation in this paper incurs heavy overhead in the thread scheduler, and hence an effective ConnP-T organization would require a more efficient interprocessor communication mechanism. A lightweight mechanism for interprocessor communication, as implemented in DragonflyBSD, would enable efficient intra-kernel messaging between processor cores. Such an efficient messaging mechanism is likely to greatly benefit the ConnP-T organization by allowing message transfer without invoking the general-purpose scheduler, and would potentially enable a ConnP-T organization to exploit the network stack cache efficiencies under heavier loads that are depicted in Figure 6.

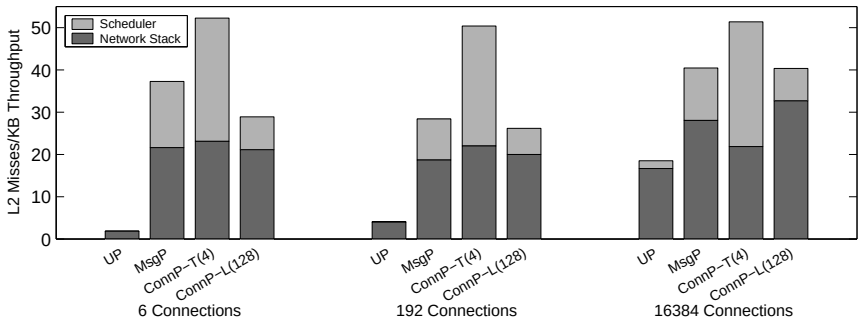

Figure 6: Profile of L2 data cache misses per 1 Kilobyte of payload data transmitted.

6 Conclusions

Network performance is increasingly important in all types of modern computer systems. Furthermore, architectural trends are pushing future microprocessors away from uniprocessor designs and toward architectures that incorporate multiple processing cores and/or thread contexts per chip. This trend necessitates the parallelization of the operating system's network stack. This paper evaluates message-based and connection-based parallelism within the network stack of a modern operating system.

The uniprocessor version of the FreeBSD operating system performs quite well, but its performance degrades as additional connections are added. Though the MsgP, ConnP-T, and ConnP-L parallel network stacks can outperform the uniprocessor when using 4 cores, none of these organizations approach perfect speedup. This is caused by the higher locking overhead, poor cache efficiency, and high scheduling overhead of the parallel organizations. While MsgP can outperform a uniprocessor by 31% on average and by 62% for loads utilizing 16384 simultaneous connections, the enormous locking overhead incurred by such an approach limits its performance and prevents it from saturating available network resources. In contrast, ConnP-T eliminates intra-stack locking completely by using thread serialization but incurs significant scheduling overhead that limits its performance to less than that of the uniprocessor kernel for all but the heaviest connection loads. ConnP-L mitigates the locking overhead of MsgP, by grouping connections to reduce global locking, and the scheduling overhead of ConnP-T, by using the requesting thread for network processing rather than invoking a network protocol thread. This results in good performance across a wide range of connections, delivering 5440 Mb/s for moderate connection loads and achieving a 126% improvement over the uniprocessor kernel when handling 16384 simultaneous connections.

Acknowledgments

This work is supported in part by the NSF under Grant No. CCF-0546140, by the Texas Advanced Technology Program under Grant No. 003604-0052-2003, and by donations from AMD.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Berger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[2] M. Björkman and P. Gunningberg. Performance modeling of multiprocessor implementations of protocols. *IEEE/ACM Transactions on Networking*, June 1998.

[3] K. Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, Oct. 1999.

[4] J. Hurwitz and W. Feng. End-to-end performance of 10-gigabit Ethernet on commodity systems. *IEEE Micro*, Jan./Feb. 2004.

[5] S. Kapil, H. McGhan, and J. Lawrendra. A chip multithreaded processor for network-facing workloads. *IEEE Micro*, Mar./Apr. 2004.

[6] H. Kim and S. Rixner. Performance characterization of the FreeBSD network stack. Technical Report TR05-450, Rice University Computer Science Department, June 2005.

[7] H. Kim and S. Rixner. TCP offload through connection handoff. In *Proceedings of EuroSys*, Apr. 2006.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, Mar./Apr. 2005.

[9] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2), 2003.

[10] K. Krewell. UltraSPARC IV mirrors predecessor. *Microprocessor Report*, Nov. 2003.

[11] K. Krewell. Double your Opterons; double your fun. *Microprocessor Report*, Oct. 2004.

[12] K. Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, Sept. 2004.

[13] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[14] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[15] V. Roca, T. Braun, and C. Diot. Demultiplexed architectures: A solution for efficient STREAMS-based communication stacks. *IEEE Network*, July 1997.

[16] D. C. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of the INFOCOM Conference on Computer Communications*, Apr. 1995.

[17] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. Power4 system architecture. *IBM Journal of Research and Development*, Jan. 2002.

[18] S. Tripathi. FireEngine—a new networking architecture for the Solaris operating system. White paper, Sun Microsystems, June 2004.

[19] R. N. M. Watson. Introduction to multithreading and multiprocessing in the FreeBSD SMPng network stack. In *Proceedings of EuroBSDCon*, Nov. 2005.

[20] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996.