This paper originally appeared at USENIX 2006

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 use concurrent threads to carry out network operations on independent messages (usually packets), whereas connection-parallel stacks map operations to groups of connections and permit concurrent processing on independent connection groups. 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.

1 Introduction

As network bandwidths continue to increase at an exponential pace, the performance of modern network stacks must keep pace in order to efficiently utilize that bandwidth. In the past, exponential gains in microprocessor performance have always enabled 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 to make up for the loss in performance growth of individual processor cores. 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 [16], 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).

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 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

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

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 for large connection loads.

The following section further motivates the need for parallelized network stacks and discusses prior work. Section 3 then describes the parallel network stack architectures. Section 4 presents and discusses the results. Finally, Section 5 concludes the paper.

2 Background

Traditionally, uniprocessors have not been able to saturate the network with the introduction of each new Ethernet bandwidth generation, but exponential gains in uniprocessor performance have always allowed processing power to catch up with network bandwidth. However, 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, but it is also difficult to further improve the efficiency of complex modern uniprocessor architectures.

To further increase performance despite these challenges, industry has turned to single chip multiprocessors (CMPs) [12]. IBM, Sun, AMD, and Intel have all released dual-core processors [2, 15, 4, 8, 9]. Sun's Niagara is perhaps the most aggressive example, with 8 cores on a single chip, each capable of executing four threads of control [7, 10]. However, a CMP trades uniprocessor performance for additional processing cores, which should collectively deliver higher performance on parallel workloads. Therefore, the network stack will have to be parallelized extensively in order to saturate the network with modern microprocessors.

While modern operating systems exploit parallelism by allowing multiple threads to carry out network operations concurrently in the kernel, supporting this parallelism comes with significant cost [1, 3, 11, 13, 18]. For example, uniprocessor Linux kernels deliver 20% better end-to-end throughput over 10 Gigabit Ethernet than multiprocessor kernels [3].

In the mid-1990s, two forms of network processing parallelism were extensively examined: message-oriented and connection-oriented parallelism. Using message-oriented parallelism, messages (or packets) may be processed simultaneously by separate threads, even if those messages belong to the same connection. Using connection-oriented parallelism, messages are grouped according to connection, allowing concurrent processing of messages belonging to different connections.

Nahum *et al.* first examined message-oriented parallelism within the user-space *x*-kernel utilizing a simulated network device on an SGI Challenge multiprocessor [11]. This study found that finer grained locking around connection state variables generally degrades performance by introducing additional overhead and does not result in significant improvements in speedup. Rather, coarser-grained locking (with just one lock protecting all TCP state) performed best. They furthermore found that careful attention had to be paid to thread scheduling and lock acquisition ordering on the inbound path to ensure that received packets were not reordered during processing.

Yates *et al.* later examined a connection-oriented parallel implementation of the *x*-kernel, also utilizing a simulated network device and running on an SGI Challenge [18]. They found that increasing the number of threads to match the number of connections yielded the best results, even far beyond the number of physical processors. They proposed using as many threads as were supported by the system, which was limited to 384 at that time.

Schmidt and Suda compared message-oriented and connection-oriented network stacks in a modified version of SunOS utilizing a real network interface [14]. They found that with just a few connections, a connection-parallel stack outperforms a message-parallel one. However, they note that context switching increases significantly as connections (and processors) are added to the connection-parallel scheme, and that synchronization cost heavily affects the efficiency with which each scheme operates (especially the message-parallel scheme).

Synchronization and context-switch costs have changed dramatically in recent years. The gap between memory system and processing performance has become much greater, vastly increasing synchronization cost in terms of lost execution cycles and exacerbating the cost of context switches as thread state is swapped in memory. Both the need to close gap between Ethernet bandwidth and microprocessor performance and the vast changes in the architectural characteristics that shaped prior parallel network stack analyses motivate a fresh examination of parallel network stack architectures on modern parallel hardware.

3 Parallel Network Stack Architectures

Despite the conclusions of the 1990s, no solid consensus exists among among modern operating system developers regarding efficient, scalable parallel network stack design. Current versions of FreeBSD and Linux incorporate variations of message parallelism within their network stacks. Conversely, the network stack within Solaris 10 incorporates a variation of connection-based parallelism [16], as does DragonflyBSD. Willmann *et al.* present a detailed description of parallel network stack organizations, and a brief overview follows [17].

3.1 Message-based Parallelism (MsgP)

Message-based parallel (MsgP) network stacks, such as FreeBSD, allow multiple threads to operate within the network stack simultaneously and permit these various threads to process messages independently. 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 carry out the requested service within the kernel. When the network interface card (NIC) interrupts the host, the NIC's associated inbound protocol thread services the NIC and processes received packets "up" through the network stack.

Given these concurrent application and inbound protocol threads, 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 locks when accessing shared connection state, such as the global connection hashtable lock (for looking up TCP connections) and per-connection locks (for both socket state and TCP state). This locking organization enables concurrent processing of different messages on the same connection.

Note that the inbound thread configuration described is not the FreeBSD 7 default. Normally parallel driver threads service each NIC and then hand off inbound packets to a single worker thread. That worker thread then processes the received packets "up" through the network stack. The default configuration limits the performance of MsgP, so is not considered in this paper. The thread-per-NIC model also differs from the message-parallel organization described by Nahum *et al.* [11], 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, hence it 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 a protocol processing thread (ConnP-T), and the second assigns each connection to a lock (ConnP-L).

3.2.1 Thread Serialization (ConnP-T)

Connection-based parallelism using threads utilizes several kernel threads dedicated to protocol processing, each of which is assigned a subset of the system's connections. At each entry point into the TCP/IP protocol stack, a request for service is enqueued for the appropriate protocol thread based on the TCP connection. Later, the protocol threads, which only carry out TCP/IP processing and are bound to a specific CPU, dequeue requests and process them appropriately. Because connections are uniquely and persistently assigned to a specific protocol thread, no per-connection state locking is required. These protocol threads implement both synchronous operations, for applications that require a return code, and asynchronous operations, for drivers that simply enqueue packets and then continue servicing the NIC.

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. When the entire tuple is not yet defined (e.g., prior to port assignment during a `listen()` call), the corresponding operation executes on protocol thread 0 and may later migrate to another thread when the tuple becomes fully defined.

3.2.2 Lock Serialization (ConnP-L)

Connection-based parallelism using locks also separates connections into groups, but each group is protected by a single lock, rather than only being processed by a single thread. As in connection-based parallelism using threads, application threads entering the kernel for network service and driver threads passing up received packets both classify each request to a particular connection group. However, application threads then acquire the lock for the group associated with the given connection and then carry out the request with private access to any group-wide structures (including connection state). For inbound packet processing, the driver thread classifies each inbound packet to a specific group, acquires the group lock associated with the packet, and then processes the packet "up" through the network stack. 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, which permits a network

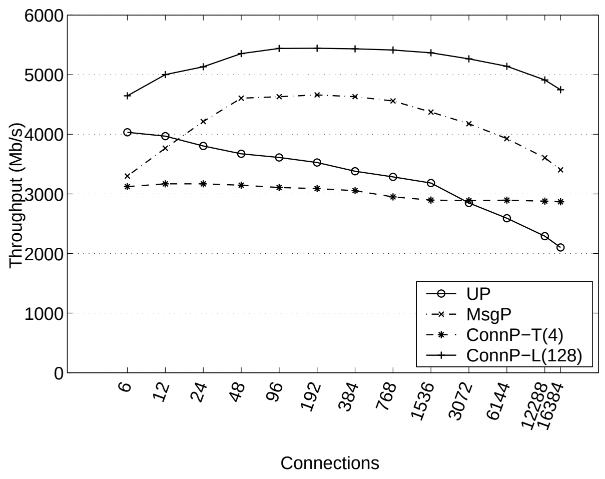

Figure 1: Aggregate network throughput.

operation to either be carried out directly after acquisition of a group lock or to be passed on to a worker thread for later processing. ConnP-L is more rigidly defined; application and inbound protocol threads always acquire exclusive control of the group lock.

4 Evaluation

The three competing parallelization strategies are implemented within the 2006-03-27 repository version of the FreeBSD 7 operating system for comparison on 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 system and three client systems. The clients never limit the network performance of any experiment.

Each network stack organization is evaluated using a custom multithreaded, event-driven TCP/IP microbenchmark that distributes traffic across a configurable number of connections and uses zero-copy I/O. This benchmark manages connections using as many threads as there are processors. All experiments use the standard 1500-byte maximum transmission unit, and sending and receiving socket buffers are 256 KB each.

Figure 1 depicts the aggregate throughput across all connections when executing the parallel TCP benchmark utilizing various configurations of FreeBSD 7. "UP" is the uniprocessor version of the FreeBSD kernel running on a single core of the Opteron server; all other kernel configurations use all 4 cores. "MsgP" is the multiprocessor MsgP kernel described in Section 3.1. MsgP uses a lock per connection. "ConnP-T(4)" is the multiprocessor 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 ConnP-L kernel described in Section 3.2.2. ConnP-L(128) divides the connections among 128 locks within the TCP/IP stack.

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 that do not succeed immediately.

The figure shows that the uniprocessor 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.

Figure 1 shows that using 4 cores, ConnP-L(128) and MsgP outperform the uniprocessor FreeBSD 7 kernel for almost all connection loads. However, the speedup 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.

4.1 Locking Overhead

Both lock latency and contention are significant sources of overhead within parallelized network stacks. Within the network stack, there are both global and individual locks. Global locks protect hash tables that are used to access individual connections, and individual locks protect only one connection. A thread must acquire a global

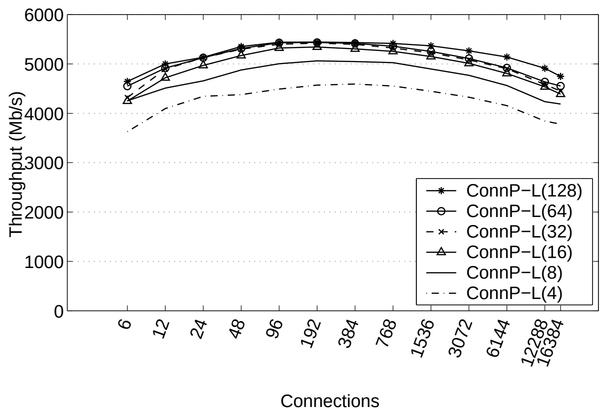

Figure 2: Aggregate network throughput for ConnP-L as the number of locks is varied.

lock to look up and access an individual lock. During contention for these global locks, other threads are blocked from entering the associated portion of the network stack, limiting parallelism.

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. The MsgP network stack experiences significant contention for global TCP/IP locks. The `Connection Hashtable` lock protecting individual `Connection` locks is particularly problematic. Lock profiling shows that contention for `Connection` locks decreases with additional connections, but that the cost for contention for these locks increases because as the system load increases, they are held longer. Hence, when a `Connection` lock is contended (usually between the kernel's inbound protocol thread and an application's sending thread), a thread blocks longer holding the global `Connection Hashtable` lock, preventing other threads from making progress.

Whereas the MsgP stack relies on repeated acquisition of the `Connection Hashtable` and `Connection` locks, ConnP-L stacks can also become bottlenecked if a single connection group becomes highly contended. Table 1 shows the contention for the `Network Group` locks for ConnP-L stacks as the number of network groups is varied. Though ConnP-L(4)'s `Network Group` lock contention is high at over 50% for all connection loads, increasing the number of groups to 128 reduces contention from 52% to just 2% for the heaviest load. Figure 2 shows the effect that increasing the number of network groups has on aggregate throughput. As is suggested by reduced `Network Group` lock contention, throughput generally increases as groups are added, although with diminishing returns.

OS Type	6 conns	192 conns	16384 conns
UP	1.83	4.08	18.49
MsgP	37.29	28.39	40.45
ConnP-T(4)	52.25	50.38	51.39
ConnP-L(128)	28.91	26.18	40.36

Table 2: L2 Data cache misses per KB of transmitted data.

OS Type	6 conns	192 conns	16384 conns
UP	481.77	440.20	422.84
MsgP	2904.09	1818.22	2448.10
ConnP-T(4)	3487.66	3602.37	4535.38
ConnP-L(128)	2135.26	923.93	1063.65

Table 3: Cycles of scheduler overhead per KB of transmitted data.

4.2 Cache Behavior

Table 2 shows the number of L2 data cache misses per KB of payload data transmitted, effectively normalizing cache hierarchy efficiency to network bandwidth. The uniprocessor kernel incurs very few cache misses relative to the multiprocessor configurations because of the lack of migration. As connections are added, the associated increase in connection state stresses the cache and directly results in increased cache misses [5, 6].

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. 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.

4.3 Scheduler Overhead

The ConnP-T kernel trades the locking overhead of the ConnP-L and MsgP kernels for scheduling overhead. Network operations for a particular connection must be scheduled onto the appropriate protocol thread. Figure 1 showed that this results in stable, but low total bandwidth as connections scale for ConnP-T. Conversely, ConnP-L minimizes lock contention with additional groups and reduces scheduling overhead since messages are not transferred to protocol threads. This results in consistently better performance than the other parallel organizations.

Table 3 shows scheduler overhead normalized to network bandwidth, measured in cycles spent managing the scheduler and scheduler synchronization per KB of payload data transmitted. 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. This highlights that ConnP-T's thread-based serialization requires efficient inter-thread communication to be effective. 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.

5 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. Further results and analysis are available in a technical report [17].

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 the heaviest connection loads, 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 intrastack 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 large connection loads.

References

[1] BJÖRKMAN, M., AND GUNNINGBERG, P. Performance modeling of multiprocessor implementations of protocols. *IEEE/ACM Transactions on Networking* (June 1998).

[2] DIEFENDORFF, K. Power4 focuses on memory bandwidth. *Microprocessor Report* (Oct. 1999).

[3] HURWITZ, J., AND FENG, W. End-to-end performance of 10-gigabit Ethernet on commodity systems. *IEEE Micro* (Jan./Feb. 2004).

[4] KAPIL, S., MCGHAN, H., AND LAWRENDRA, J. A chip multithreaded processor for network-facing workloads. *IEEE Micro* (Mar./Apr. 2004).

[5] KIM, H., AND RIXNER, S. Performance characterization of the FreeBSD network stack. Tech. Rep. TR05-450, Rice University Computer Science Department, June 2005.

[6] KIM, H., AND RIXNER, S. TCP offload through connection handoff. In *Proceedings of EuroSys* (Apr. 2006).

[7] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro* (Mar./Apr. 2005).

[8] KREWELL, K. UltraSPARC IV mirrors predecessor. *Microprocessor Report* (Nov. 2003).

[9] KREWELL, K. Double your Opterons; double your fun. *Microprocessor Report* (Oct. 2004).

[10] KREWELL, K. Sun's Niagara pours on the cores. *Microprocessor Report* (Sept. 2004).

[11] NAHUM, E. M., YATES, D. J., KUROSE, J. F., AND TOWSLEY, D. Performance issues in parallelized network protocols. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (Nov. 1994).

[12] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 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).

[13] ROCA, V., BRAUN, T., AND DIOT, C. Demultiplexed architectures: A solution for efficient STREAMS-based communication stacks. *IEEE Network* (July 1997).

[14] SCHMIDT, D. C., AND SUDA, T. Measuring the performance of parallel message-based process architectures. In *Proceedings of the INFOCOM Conference on Computer Communications* (Apr. 1995).

[15] TENDLER, J. M., DODSON, J. S., J. S. FIELDS, J., LE, H., AND SINHAROY, B. Power4 system architecture. *IBM Journal of Research and Development* (Jan. 2002).

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

[17] WILLMANN, P., RIXNER, S., AND COX, A. L. An evaluation of network stack parallelization strategies in modern operating systems. Tech. Rep. TR06-872, Rice University Computer Science Department, Apr. 2006.

[18] YATES, D. J., NAHUM, E. M., KUROSE, J. F., AND TOWSLEY, D. 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).