Parallelism and Performance in the OpenVMS TCP/IP Kernel

Robert Rappaport, HP Software Engineer

Yanick Pouffary, HP Software Technical Director

Steve Lieman, HP Software Engineer

Mary J. Marotta, HP Information Developer

Introduction

In October 2003, TCP/IP Services for OpenVMS introduced into OpenVMS production environments a radically modified and improved Scalable Kernel. The Scalable Kernel enables parallelism in TCP/IP by taking advantage of available CPU capacity in a multiCPU configuration. It allows network performance to scale almost linearly as CPUs are added to the configuration. The Scalable Kernel was designed to enhance network application performance without jeopardizing the integrity of the basic UNIX code.

The SMP Challenge

The TCP/IP Kernel maintains a large in-memory database. Access to this database is synchronized by the use of several spin locks, all of which are associated with interrupt priority level (IPL) 8. On single CPU systems, only one active IPL 8 thread executes at a time. Therefore, there is no possibility for contention for the TCP/IP-specific spin locks on single CPU systems. On multiCPU systems, however, the potential for such contention increases as the number of CPUs in the configuration increases. The Scalable Kernel eliminates this contention.

When customers add CPUs to symmetric multiprocessing (SMP) systems, they expect the extra processing power to boost network performance, but the classic TCP/IP kernel does not take advantage of the extra processing power of the added CPUs. The number of users may actually increase, but almost all network I/O interactions are handled while holding the TCP/IP global spin lock (I/O lock 8). Contention for this lock makes it difficult to increase network throughput under these circumstances.

The Architecture of the TCP/IP Kernel

The OpenVMS TCP/IP kernel, the heart of the OpenVMS TCP/IP architecture, was ported from BSD UNIX. It was intentionally designed to operate like the UNIX- TCP/IP kernel and to interoperate with the OpenVMS operating system with a minimum of programming changes.

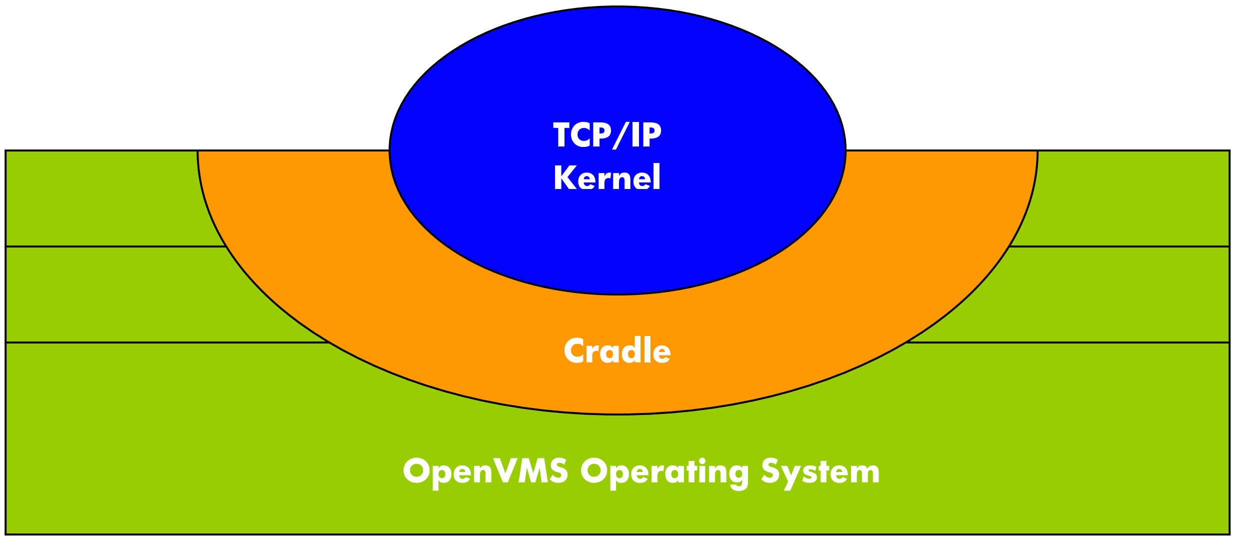

Figure 1 The Architecture of the TCP/IP Kernel

As illustrated in Figure 1, the OpenVMS TCP/IP kernel consists of two distinct parts:

- The **TCP/IP kernel** -- code that is ported from UNIX.
- The **cradle** -- OpenVMS code that supports and nurtures the UNIX code.

The cradle surrounds the UNIX code, creating an environment in which only a small percentage of the UNIX code has to be made aware that it is not operating in a UNIX system. The cradle provides transparent UNIX-like interfaces that serve the ported UNIX code in three general areas:

- User-level I/O requests are preprocessed in the cradle and fed into the UNIX code at the appropriate point.
- I/O terminations from the UNIX code are intercepted by the cradle transparently, as are all UNIX interactions with the LAN drivers.
- All interactions from the UNIX code with the OpenVMS operating system, such as the dynamic allocation and deallocation of memory, are handled transparently.

TCP/IP Thread Contexts

Code executing in the TCP/IP kernel is either in **process context** or **kernel context** mode. A thread running in process context mode has access to the user address space (for example, the user's buffers). Threads running in process context are almost always executing code in the cradle. Threads running in kernel context run at IPL 8 holding the TCP/IP global spin lock.

In the classic TCP/IP kernel, when a thread changes mode to kernel context, it has to wait for the TCP/IP global spin lock. In the Scalable Kernel environment, kernel context threads are created as IPL fork threads, which then acquire the TCP/IP global spin lock. Kernel context threads are almost always executing in the UNIX-ported portion of the code.

Figure 2 illustrates how process context threads running in the classic Kernel environment contend for I/O lock 8 (the TCP/IP global spin lock of that environment) in order to change their mode to kernel. Once a thread acquires this spin lock it can then proceed to carry out its TCP/IP kernel work while all process context threads contending for this spin lock must wait, spinning and wasting valuable CPU cycles.

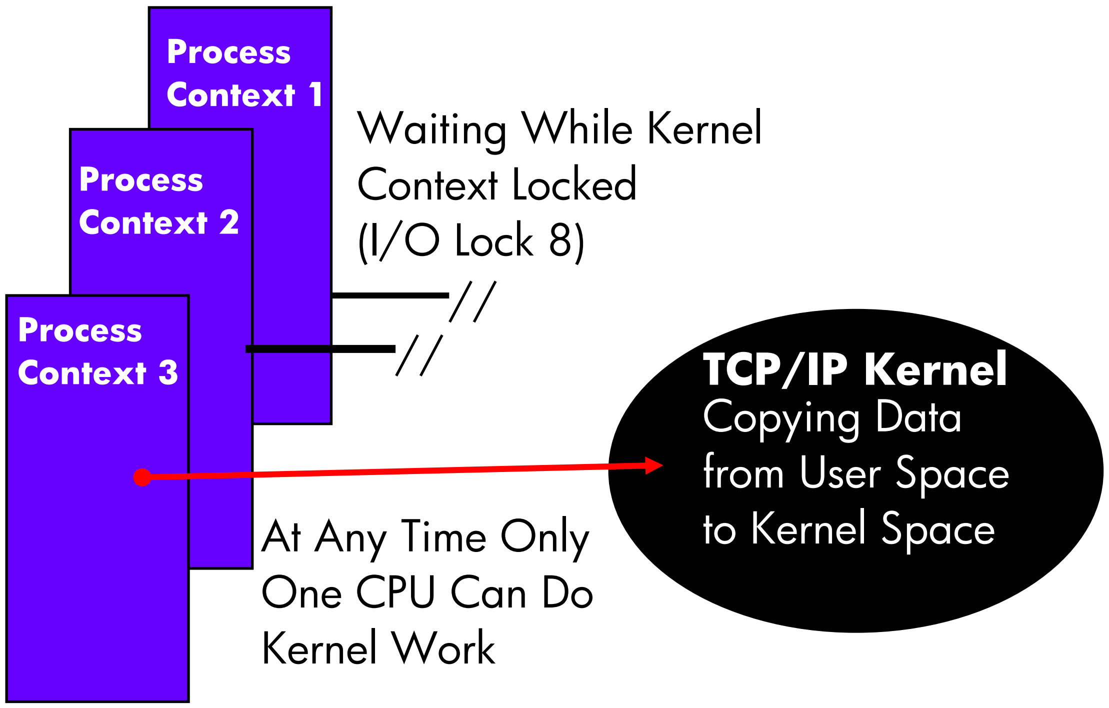
Process Context 1
Process Context 2
Process Context 3
Waiting While Kernel Context Locked (I/O Lock 8)
TCP/IP Kernel
Copying Data from User Space to Kernel Space
At Any Time Only One CPU Can Do Kernel Work

Figure 2 Process Context in the Traditional Kernel

As the number of network links per system gets larger and as the links get faster and faster, the potential number of network users requesting service can expand rapidly. As the demand increases, more and process context threads end up spinning in a loop, waiting for service while other threads are processed

Introducing Parallelism into the TCP/IP Kernel

To address the problem of wasted CPU cycles spent spinning and to allow more work to get done on SMP systems, **parallelism** was introduced into the TCP/IP kernel code. Analysis of the classic kernel showed that only a small part of the processing of network operations had to be done while the TCP/IP internal database was locked. It was possible to change the order of the code flow in the two most frequently invoked network operations (read and write) so that:

- The kernel context portion of each read or write could run in an IPL 8 fork thread.
- The completion of read and write operations would not depend on these IPL 8 fork threads being completed.

In other words, read and write operations could be designed so that the process context portion of the work queues an IPL 8 fork thread to complete the kernel context portion of the work. Once this fork thread is queued, the user I/O request can then be completed. This is how the TCP/IP Scalable Kernel works.

In the Scalable Kernel, read and write operations are processed at IPL 2 in process context and queue IPL 8 fork threads to complete the kernel context work. Because each read or write operation does not have to wait until the fork thread has completed, the operation can be marked as completed (I/O Posted) immediately after queueing the fork thread.

The IPL 8 fork threads that are operating in kernel context need to acquire the TCP/IP global spin lock in order to access the in-memory database. Allowing these fork threads to run on any available CPU would lead to contention for the spin lock. Therefore, all of these TCP/IP kernel context fork threads are directed to a queue that is processed on one specific CPU in the configuration (the designated TCP/IP CPU) on a first-come, first-served order. Because all the threads in the system that need to acquire the TCP/IP global spin lock run on one single CPU, contention for this spin lock is eliminated. And because this spin lock is no longer I/O lock 8, no other OpenVMS code will attempt to use it.

The Scalable Kernel introduces a new mechanism for code to request the creation of a kernel context thread. The mechanism involves allocating a newly-defined data structure (the TCPIP_KRP), filling in the TCPIP_KRP, and then queuing this data structure to a global work queue. If the queue is empty at the time, an IPL 8 fork thread is created, which will run on the designated TCP/IP CPU and which will process every TCPIP_KRP in the queue.

Tracking a Write Operation

The object of any TCP/IP write operation is to take data from a user buffer and place this data into a socket. The operation is performed in two distinct steps:

1. Copy the user data from the user buffer into a system buffer (MBUF) or a chain of system buffers (does not require holding the TCP/IP global spin lock)
2. Append this chain of system buffers into the socket (requires holding the TCP/IP global spin lock)

In the Scalable Kernel, the processing of a write operation is straight-forward. One or more MBUFs are allocated to accommodate the user data, and then the data is copied from user space into the new MBUF chain. A TCPIP_KRP is allocated and initialized so that it requests that this new MBUF chain be appended to the data in a particular socket. The initialization of the TCPIP_KRP includes passing the address of the MBUF chain, the address of the socket, and so forth. After the TCPIP_KRP is initialized, it is queued to the global work queue and the write request is completed.

At the same time that the write operation is being processed on one CPU, another write operation can be processed on another CPU in the system. Presumably, the other write operation is writing to a different socket. Because neither of these operations needs to acquire the global spin lock to complete, both operations run to completion without any interference. Similarly, they can run in parallel with ongoing read operations as well.

The power of the design of the Scalable Kernel becomes obvious. In a large multiCPU system, user programs running in parallel on the various CPUs of the system constantly call TCP/IP operations such as read and write. They run to completion, in parallel, without interfering with each other. Each of these requests leaves behind a TCPIP_KRP that is queued to be processed on the designated TCP/IP CPU; the processing of these TCPIP_KRP requests also runs in parallel with all the other operations.

Each process context operation leads to an associated kernel context operation. The amount of work entailed in each kernel context operation adds to the load of work on the designated TCP/IP CPU, but as long as this designated CPU is not completely saturated with work, the Scalable Kernel is able to scale close to linearly as more CPUs are added to the configuration.

The Scalable Kernel takes advantage of multiple CPUs by separating the user processes from the kernel process. Rather than blocking the CPU, it queues new user I/O requests. The flow of the send and receive logic in the cradle runs from start to finish without any interference from other TCP/IP threads. When they are successful, operations leave a pending asynchronous kernel-context thread to complete their requests. The user application does not have to wait for the kernel context thread to complete. When it queues the kernel context thread, the user request is completed. Network operations become more like transaction-oriented operations, where the parallel threads prepare transactions to be processed by the designated TCP/IP CPU.

As illustrated in Figure 3, applications no longer compete with one other to acquire locks in order to proceed.

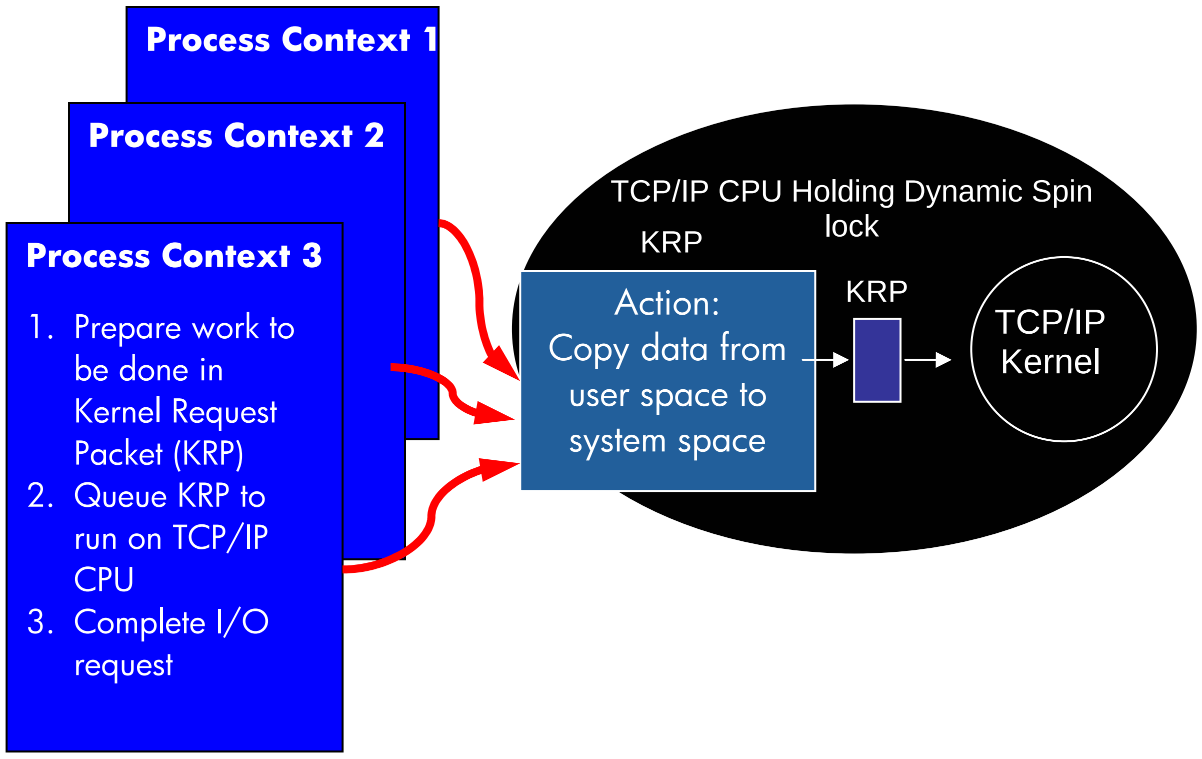
Process Context 1
Process Context 2
Process Context 3
1. Prepare work to be done in Kernel Request Packet (KRP)
2. Queue KRP to run on TCP/IP CPU
3. Complete I/O request
TCP/IP CPU Holding Dynamic Spin lock
KRP
Action: Copy data from user space to system space
KRP
TCP/IP Kernel

Figure 3 Process Context Threads in the Scalable Kernel

Types of Kernel Request Packets (KRPs)

The TCPIP_KRP describes the request to perform an operation in kernel context, including a pointer to the action routine to be called, and a series of parameters that the routine will need to complete the request. There are many different types of requests for kernel context work in the Scalable Kernel.

In total, there are over 50 different types of KRPs in the Scalable Kernel. The type of KRP created depends on the work:

- A thread executing in process context that wishes to **write** data to a socket packages up all the data to be written to the socket inside a KRP and then creates a kernel context thread to process the KRP. The processing of this KRP includes extracting the information from the KRP and calling the UNIX routines that insert new data into the transmit channel of a socket.
- A thread receiving a call from the OpenVMS LAN driver must pass **received** data from the network. This thread packages the received network data in a KRP and then creates a kernel context thread to process this KRP. To process this KRP, the kernel has to parse the received network data (IP header, TCP or UDP header, and so forth), place the parsed

data into the receive channel of a socket, and possibly wake up a user thread waiting for data to arrive on this socket.

- When a thread running a **TCP/IP timer** goes off, the information about the timer is packaged in a KRP and a kernel context thread is created to process it, executing the appropriate code to deal with the specific timer that expired.

Kernel Context Threads

The processing of kernel context threads is invisible to the TCP/IP application program. All the kernel threads access the same shared in-memory database, which holds information that cannot be accessed concurrently by more than one thread at a time. Processing in kernel context is ensured by the fact that the threads that execute in kernel context are all directed to a single, designated CPU in the configuration, where they execute one by one, at high priority and at high speed with no interference from other threads.

Instead of I/O lock 8, the Scalable Kernel uses several new dynamic spin locks, like the TCP/IP global Spin lock, which is held for relatively long periods of time, and several mini-spin locks, which are never held for very long. Each TCPIP_KRP is processed in an IPL 8 fork thread on the designated TCP/IP CPU, while holding the TCP/IP global Spin lock. Since all of the threads that need the TCP/IP global spin lock run on the TCP/IP CPU, there is never any contention for the spin lock.

Executing all the kernel threads on the same CPU also optimizes CPU cache utilization because the same objects in the shared database are usually referenced from the same CPU.

The Scalable Kernel

TCP/IP Services Version 5.4 introduces the Scalable Kernel as an optional new feature for the specific purpose of validating and quantifying the performance gains for those systems with the heaviest TCP/IP loads on SMP systems. The Scalable Kernel significantly improves the potential for performance gains depending on the applications and configuration.

The Scalable Kernel will be the default TCP/IP kernel in the next major release of TCP/IP Services for OpenVMS beyond V5.4. Tests have shown equivalent or better operational performance on single-CPU systems, and indisputable benefits for multiCPU systems under the heaviest TCP/IP loads. To obtain the benefits of the Scalable Kernel, you must upgrade your system to TCP/IP Services Version 5.4 or higher.

Measuring Throughput

The maximum gain to expect in system throughput by using the Scalable Kernel is a direct function of the amount of MPSYNCH that is attributable to TCP/IP, based on measurements using the classic TCP/IP kernel. System throughput gain is always highly application-dependent.

In a given configuration running the Scalable Kernel, the amount of remaining capacity (headroom) can be estimated by measuring the amount of time that the TCP/IP global spin lock is held by the designated TCP/IP CPU under heavy TCP/IP load. For example, in a multiCPU configuration, if the TCP/IP global spin lock is held for 40% of the time, the number of CPUs in the configuration can be doubled before causing TCP/IP bottlenecks.

Scalable Kernel Performance Tests

The following graphs show real-life data confirming the success of parallelism in the field. Note that although performance tests may show higher I/O operations per second, better utilization of system resources, and so forth, the customer is only interested in getting more of his work done in a specific unit of time.

Performance Summary

On a16-CPU GS160 Wildfire system running the classic TCP/IP kernel, the testbed application resulted in an MPSYNCH backup averaging four or more CPUs. That is, at any given time, four or more CPUs were spinning and doing nothing constructive. (This represents 25% to 35% of the potential productivity of a 16-CPU machine!)

The Scalable Kernel restores that 25% to 35% gain in throughput, virtually eliminating the waste of the previously spinning CPUs. In other words, the Scalable Kernel allows greatly expanded parallelism to make use of previously lost CPU cycles.

Comparing the Traditional Kernel to the Scalable Kernel

These test results show the overall pattern of improvement when the scalable kernel is running (in green), and when it is not running (in red).

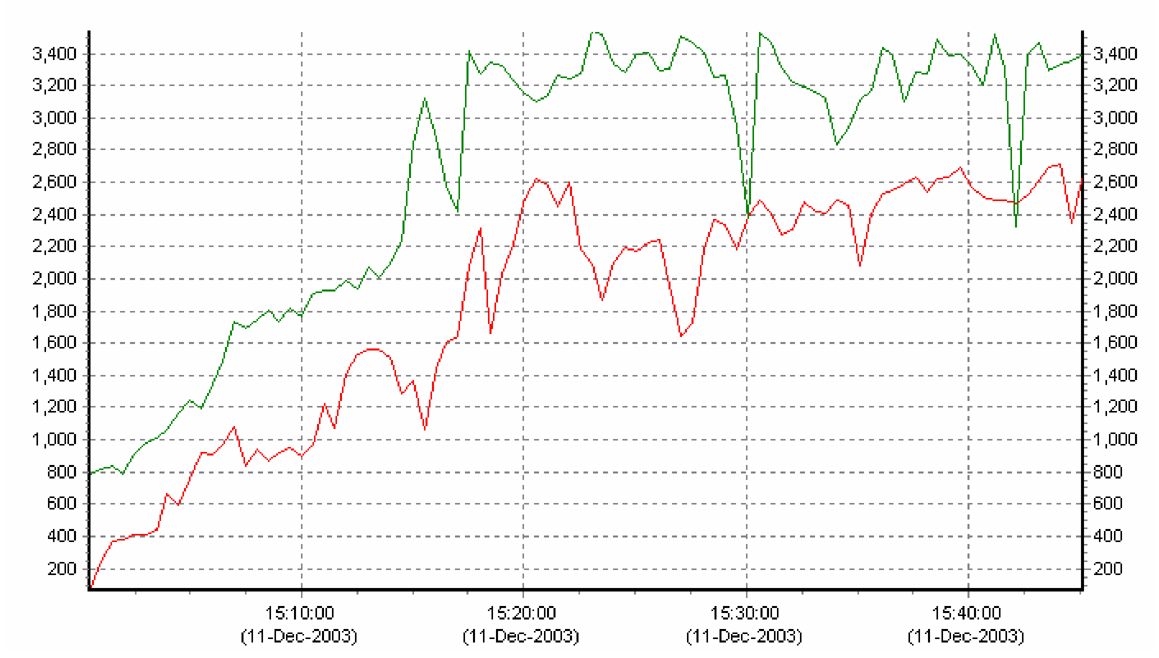

Figure 4 - TCP/IP Transmit Packets per Second

The purpose of the scalable kernel is to increase the amount of network I/O that an OpenVMS system can process. As Figure 4 shows, the rate of TCP/IP traffic can potentially increase by 30% or more when the Scalable Kernel enabled.

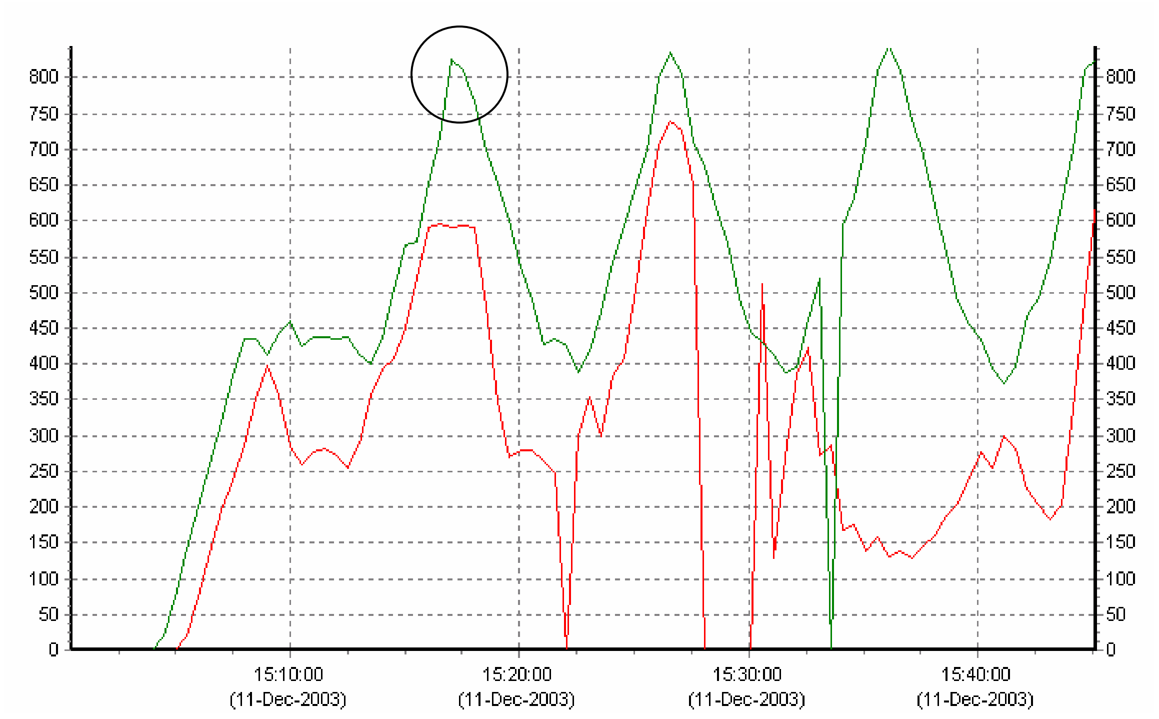

Figure 5 – Customer Orders per Minute

As shown in Figure 5, the Scalable Kernel allows the system to format more customer orders in a given time than the classic kernel. When the load gets heavy, the Scalable Kernel is able to respond and complete more real work per minute than previously possible.

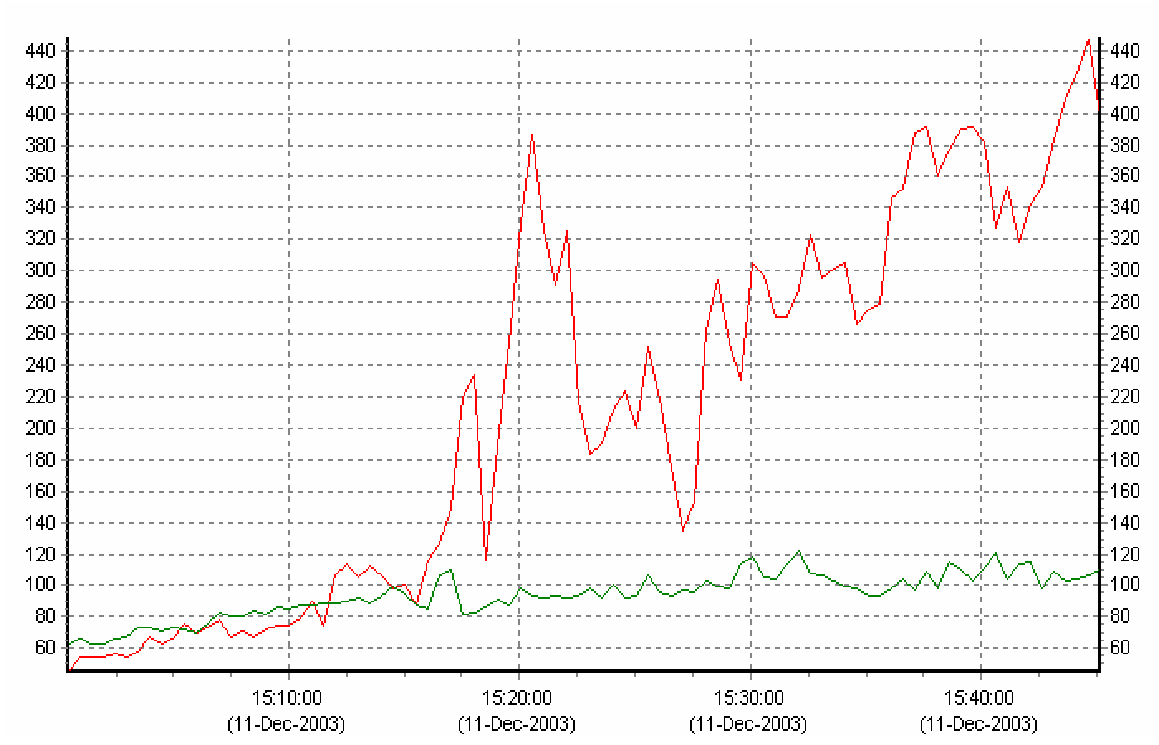

Figure 6 – MPSYNCH Percentage

When the Scalable Kernel is running, multiprocessor synchronization contention (MPSYNCH) is dramatically reduced, as shown in Figure 6.

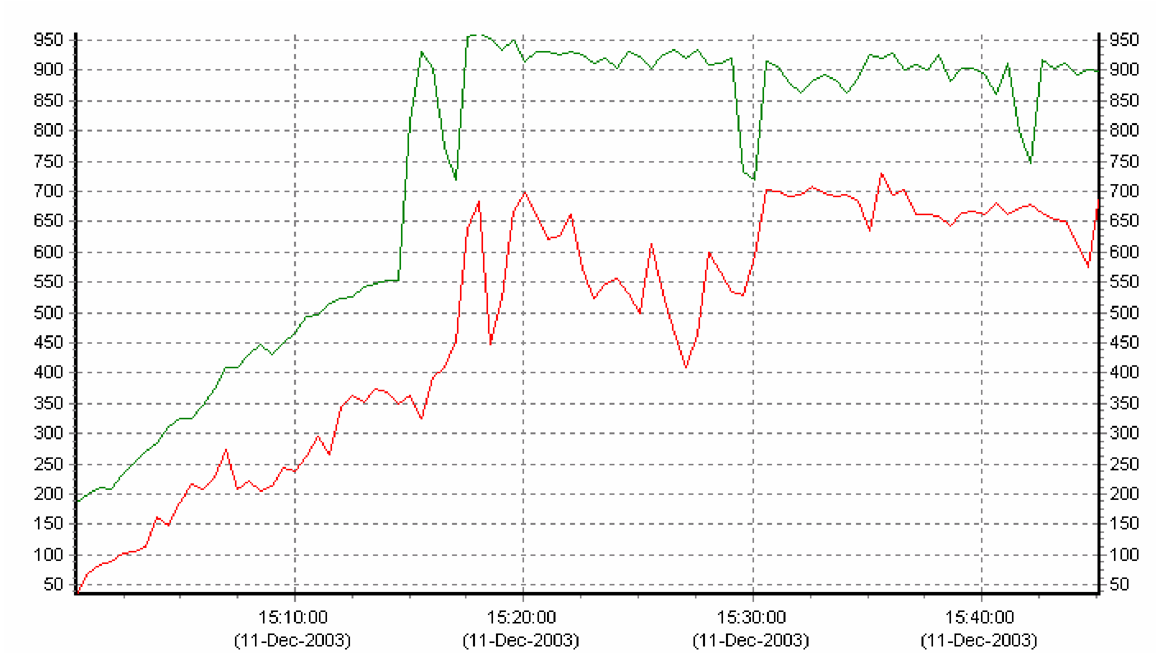

Figure 7 – Percentage of CPU Busy

As Figure 7 shows, a greater percentage of CPU time is spent in user-mode, which means that more application work is getting done when the Scalable Kernel is running.

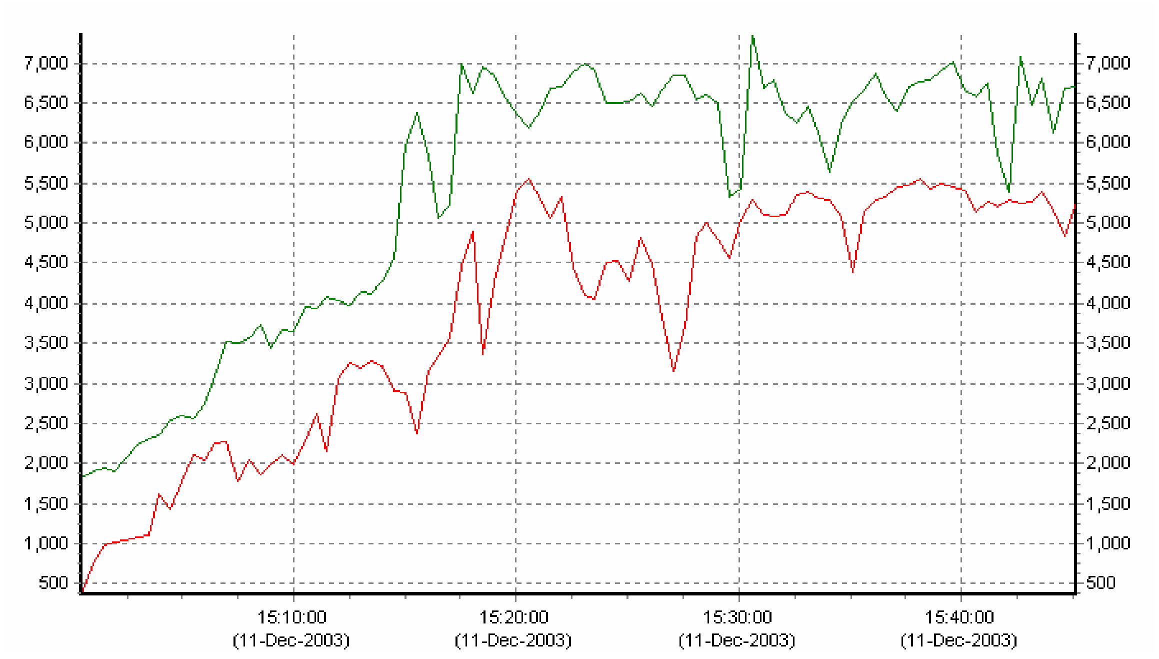

Figure 8 - Buffered I/O Rate

On OpenVMS, TCP/IP I/O activity is expressed as buffered I/O. Figure 8 shows how the rate of buffered I/O increases when the Scalable Kernel is running.

Comparing the Traditional Kernel to the Scalable Kernel

	Traditional Kernel	**Scalable Kernel**
Hold % All Locks	54.8%	35.2% (while completing more work)
Spin % All Locks	24.1%	4.4% (very good)
Locks Per Second	204,153	162,033 (doing more work with far fewer locks per unit of work)
I/O Lock 8 Hold Time	31.4%	<5% (very good. Now much more of I/O lock 8 is available for handling heavy disk I/O.)

The Importance of Maintainability

Over half the OpenVMS TCP/IP kernel code is ported from UNIX and the TCP/IP code base is under constant development. In order for the OpenVMS TCP/IP kernel to remain up-to-date with leading-edge functionality, frequent infusions of new UNIX code are required. The OpenVMS TCP/IP engineering team must repeat the port of the UNIX code periodically. The amount of OpenVMS modifications introduced into the ported UNIX code must be restricted, so that re-porting operations remain a manageable task that can be accomplished in a time period of weeks, not years.

Limits on the amount of changes that could be introduced into the UNIX code dictated the approach to the challenge of achieving parallelism in the TCP/IP Services for OpenVMS product. Complicated locking schemes that would require that lock domains span both the ported UNIX code and the OpenVMS cradle would have greatly increased the complexity of the solution, introducing issues of quality as well as increased maintenance.

The Scalable Kernel is the ideal solution because it is customized to the OpenVMS SMP environment, it operates just as well in a single-CPU configuration as an SMP system, and it imposes the least amount of overhead for future maintenance.

Future Kernel Enhancements

In the future, the Scalable Kernel may be enhanced to handle even larger CPU configurations. To accomplish this, the current single, shared in-memory database could be divided into two or more databases, each of which would be serviced by its own designated kernel context CPU. This would ensure that the designated TCP/IP CPU does not become a limit to system throughput.

Additional performance gains can be realized by optimizing the processing of the transactions, so that the designated TCP/IP CPU takes less time and effort to process each individual transaction, thereby supporting a greater number of parallel threads without becoming overloaded.

For More Information about TCP/IP Services Performance

The Scalable Kernel allows greater parallelism in the processing of TCP/IP requests and is not a generalized performance panacea that solves everything. Processing tens of thousands of TCP/IP packets and distributing them over thousands of sockets requires CPU cycles, which will inevitably take its toll. The Scalable Kernel allows you to deal efficiently with this necessary use of CPU resources by adding CPUs to the configuration and then allowing these additional CPUs to be effectively used by the system instead of merely spinning, doing nothing and getting in the way.

There are other steps, independent of the Scalable Kernel, which you can take to improve the performance of individual TCP/IP operations, including TCP/IP tuning, adjusting window sizes for sockets, and so forth. For more information about these performance enhancement techniques, consult the *TCP/IP Services for OpenVMS Tuning and Troubleshooting* guide.