VTRACE and Communication Performance Analysis

Peng Dai and Thomas W. Doeppner Jr.

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-36
November 1995

VTRACE and Communication Performance Analysis

Peng Dai[†]
Thomas W. Doeppner Jr.[†]

Department of Computer Science
Brown University
Providence, RI 02912
November 1, 1995

Abstract

As the requirements for communication performance grow and the vast variation and improvement in hardware components increase, the various functionally different software layers have been increasingly responsible for performance degradation. In this paper, we explore the use of VTRACE, a tracing facility provided in Solaris, as a general methodology in communication performance analysis and also present performance data for an experimental version of Sun's Solaris Operating System.

1. Introduction

With the interrelated technologies of the world-wide web, multimedia, and computer clustering becoming increasingly popular, communication performance is rapidly attaining a major role in application performance. Many applications have stringent bandwidth and latency requirements. Modern communication technologies can provide the raw bandwidth and responsiveness needed to meet these requirements, but the communications software running on the communicating computers often cannot cope — it was designed to handle smaller amounts of data more slowly and often needs redesign to deal with current requirements.

The speed at which processors execute instructions is increasing exponentially over time. The speed at which communication hardware can transfer data is increasing at a similar rate. Data transmitted and received over a communication line by a computer must be processed by software implementing the communication protocol before it is made available to user applications. A major concern is the expense of this processing. I.e., there might be so much time spent on protocol processing that there is a marked difference between bandwidth and latency as measured between network interfaces than as measured between user applications. This difference, the protocol overhead, could exist, despite the fact that instruction execution speeds have increased so dramatically, because of the costs of moving data within primary storage as the data is processed by the various layers of software.

It is certainly easy to determine if protocol overhead is significant: one simply compares actual communication performance with the rated performance of the communication hardware. Determining the extent of the problem and what, if anything, can be done about it requires much more work. Protocols are organized into layers; we need to measure the overhead of each layer and that of communicating data between layers. Our only means for measuring this overhead is to add instructions to record the

[†]This work was supported by a grant from Sun Microsystems, Inc.

times of events such as entering and exiting such layers. However, recording this information carries its own overhead, which, if a large number of measurements are taken, can be significant. Furthermore, the code paths taken with the protocol code may have timing dependencies: the extra time spent to record an event may cause the subsequent code path to be different (perhaps requiring more time, perhaps less) than it would have been otherwise.

In this paper we discuss our use of Solaris's VTRACE facility to examine the performance of communication protocols. We have run a number of experiments involving the transfer of data between machines and have used VTRACE to produce performance data. We have developed tools that both extract and summarize information relevant to our experiments from the huge amount of raw data produced by VTRACE. With the help of these tools, we analyzed the costs of the various software components that are involved in communication.

Section 2 of this document discusses the VTRACE facility in general, and explores the pros and cons of using it for performance measurement purposes and the possibility of automating the data analysis. Section 3 provides a high-level overview of the Solaris communication design, including the STREAMS framework and where the various protocol stacks and network driver modules fit, that prepares a soft landing into data analysis.

Section 4 discusses the test methodology and test configuration, and introduces the conventions used for data analysis. The performance data of various protocol stacks and network drivers are presented as well. Note that the data presented here were obtained on an experimental version of Solaris and should not be used to infer the performance of any versions of Solaris in production.

Section 5 discusses the details of verifying VTRACE methodology using the performance data obtained in Section 4, and section 6 summarizes our results and proposes future directions for related research.

2. *VTRACE Methodology*

This section covers the following aspects of VTRACE methodology:

- General mechanism
- Potential sources of perturbation
- Proposal for rectification of perturbation
- Verification
- Data collection and analysis

2.1. *VTRACE in a nutshell*

VTRACE is a general tracing mechanism used to trace both kernel-level and user-level code (see /usr/include/sys/vtrace.h for details). VTRACE models the system by a collection of CPUs, a collection of threads, a collection of code modules mapped into memory that are to be executed and the time of execution at a specific location within a module. The CPUs are identified by CPU numbers, threads by thread ids, time by system clock ticks and locations within the code modules by system-wide unique identifiers. Each of these elements specifies a dimension in the VTRACE space.

One can think of the execution history of a thread as a life trajectory in VTRACE space, starting from the creation of the thread and marching through relevant code modules until death. Along the life trajectory, the thread can be scheduled on any CPU at any time and executes a number of code modules at different times.

VTRACE allows real-time sampling of the trajectory in a user-definable way: users can specify where in the code to sample. The sampled point on the trajectory must contain the CPU number, current time and code location for the sampled thread. Given all these coordinates, the behavior of the thread is then completely specified, as illustrated in Figure 1.

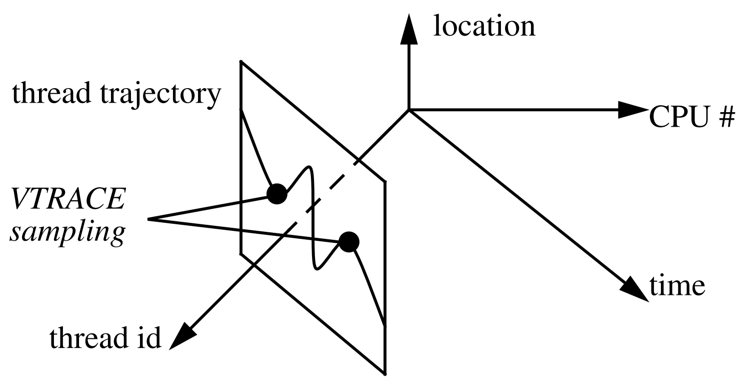

FIGURE 1. VTRACE Space

Next let's introduce some terminology.

2.1.1. VTRACE Event Space

A VTRACE event is defined as a specific location within the code. All possible events constitute the event space. An event can be further broken down into a facility field and a tag field. The former partitions the event space into functionally orthogonal subspaces, e.g., SYSCALL, TCP, NFS. Of course the location that an event identifies is determined by the implementation: once it is determined, it remains there. Thus, a VTRACE event is a static concept determined at compile time. VTRACE allows VTRACE events to be enabled or disabled either by facility or by individual event. A disabled VTRACE event is sampled.

2.1.2. VTRACE Point

A VTRACE point is a point along the life trajectory of a thread (Figure 1). Not all points along the trajectory are VTRACE points, only potentially sampled points are, including both enabled and disabled VTRACE points. A VTRACE point consists of a VTRACE event, current time, CPU number, thread id and possibly other information, and uniquely defines a position within VTRACE space.

What happens when a thread hits a VTRACE point? Or, to put it another way, how does a VTRACE point get recorded? As shown in Figure 2, the per-CPU VTRACE event map, which contains control information for each VTRACE event for the current CPU, is checked to see whether the VTRACE event of the current hit VTRACE point is enabled. If so, it branches to a specific function that records the VTRACE point of the current hit in a per-CPU VTRACE buffer. Otherwise, it does nothing. The macro `TRACE_N` defined at the end of `/usr/include/sys/vtrace.h` does all this.

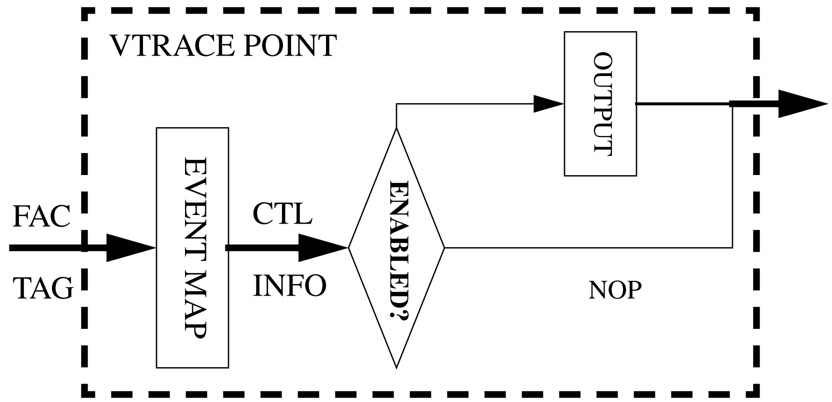

FIGURE 2. Hitting a VTRACE Point

2.1.3. VTRACE Record

Upon hitting a VTRACE point, certain information is recorded for further identification of the location in VTRACE space. This information is encapsulated in a VTRACE record (Figure 3).

A VTRACE record contains a header word followed by several fields of either integer or string format. The header word itself contains the current VTRACE event and time.

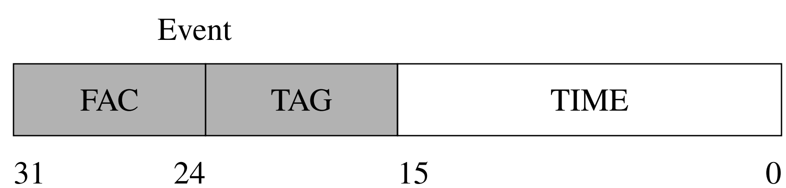

FIGURE 3. VTRACE Record Header

Preceding the very first occurrence of a VTRACE record of a certain VTRACE event, a VTRACE label record (Figure 4) is output once and for all that contains the VTRACE record header, VTRACE facility, VTRACE tag, the length, name and format of the record and a bitmap indicating which fields in the format string are numbers and which are strings.

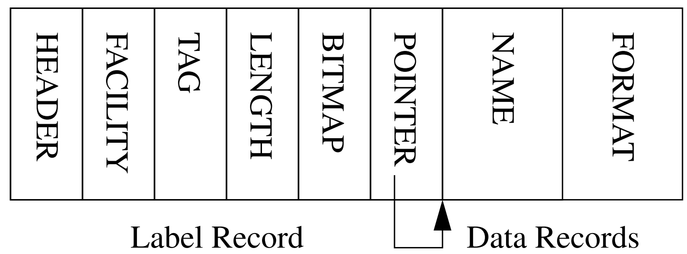

FIGURE 4. VTRACE Label Record Layout[1]

Each succeeding VTRACE point consults the VTRACE label record to fill the data into the format string.

Separate VTRACE records describe the CPU number and thread id that are output only when they are changed. This eliminates the needs for a CPU number and thread id field in other VTRACE records.

VTRACE records bear a record type indicating the number of data fields and whether they are integers or strings. Each record type has a different cost.

2.1.4. VTRACE File

As mentioned above, that there is a VTRACE buffer for each CPU and each enabled VTRACE point writes out a VTRACE record in chronological order into the buffer belonging to the CPU where the thread hitting the VTRACE point is scheduled. But how do we get a single VTRACE file out of these VTRACE buffers?

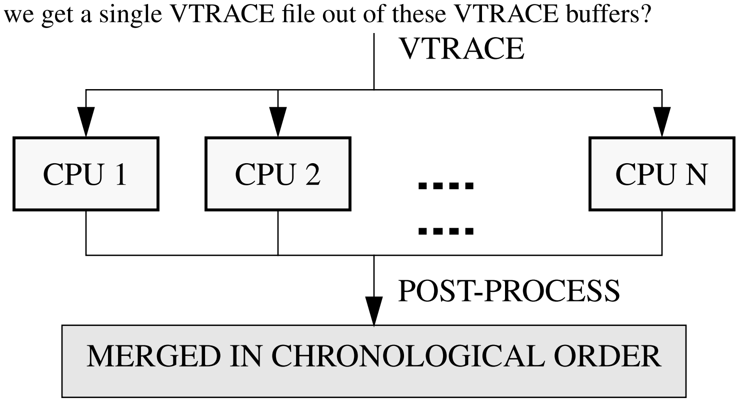

FIGURE 5. VTRACE File Generation

As Figure 5 shows, VTRACE file has two sections, both of which are paged. Each page starts with the administrative VTRACE records such as total elapsed time record, writing thread id and CPU number records, and ends with padding records. In addition to the standard administrative VTRACE records on each page, the administrative section also contains VTRACE information such as version record, label records, starting time record, absolute time record, page size record, total number of

1. Pointers are used in VTRACE to handle variable-length strings.

CPUs record, current CPU and thread records, and title record. The data section contains non-administrative VTRACE records.

2.2. VTRACE Perturbation

The goal of any measurement is to extract relevant information from the target. Thus, interaction and perturbation are inevitable between the measurer and the measured. This observation applies to VTRACE as well. In this section we take a look at where the perturbations enter and how they can be rectified. Finally, we briefly go through some data to get a feeling for how objective VTRACE is and to what extent it reflects the truth. Details of the verification are discussed in section 5.

2.2.1. Sources of Perturbations

Almost all VTRACE sampling can be classified into three categories:

- Enabled VTRACE Points

This is a simple but relatively strong source of perturbation. The cost of such a VTRACE point consists of the overhead of testing the status, i.e., enabled or disabled, and writing out the VTRACE record:

$$C_E = C_T + C_R \tag{1}$$

where C_E is the total cost of an enabled VTRACE point, C_T is the overhead and C_R is the cost of writing the VTRACE record.

Different VTRACE records have different weight, so that C_E is a function of VTRACE record type. Generally, C_R is much larger than C_T.

- Disabled VTRACE Points

This is a simple source of perturbation, smaller than the previous one. The overhead of a disabled VTRACE point is solely that of testing the status; the record is not written in this case:

$$C_D = C_T \tag{2}$$

where C_D is the total cost of an disabled VTRACE point and C_T is as above.

- Code-path Variation

This kind of perturbation is inherently much more subtle (it exhibits the fingerprint of chaos, as is not surprising). The idea here is that the execution history, or simply the code path, is significantly distorted under VTRACE. Figuratively, the effect of sampling at a single point of the life trajectory cannot be absorbed by a local distortion, but instead manifests itself globally by relentlessly collapsing the entire structure of the trajectory.

There are two cases in which this could happen. First, it could be the case that a code path that is periodic and regular before VTRACE is added becomes chaotic and unpredictable afterwards. Second, the code path could already be irregular and radical even before VTRACE comes into play. The first case corresponds to a critical phenomenon whose probability is negligible; in the second case, although the code path has been altered, the fact that it is intrinsically chaotic has not been changed.

In real systems, load is an indirect measure of the degree of irregularity. Heavy load can give rise to some complicated interactions among subsystems that are responsible for observed chaotic behavior. Analyzing performance of heavily loaded and therefore chaotic systems is, however, beyond the scope of this paper.

2.2.2. Rectifying VTRACE Perturbation

Although we cannot avoid VTRACE perturbations, we can rectify them so that our results are objective.

- Enabled VTRACE Points

The designers of VTRACE have foreseen the perturbation problem and provided a special facility called SPEED to obtain the cost of outputting various kinds of VTRACE records, or C_R. A command-line interface is also provided so that one can actually get the cost right out of the box. Using this, one can easily rectify most of the effects of all enabled VTRACE points in the following way.

Suppose that we want to derive the cost of the code bounded by two VTRACE points and that the code goes through a number of other enabled VTRACE points. We simply count the intermediary enabled VTRACE points by VTRACE record type and calculate a subtotal for each VTRACE record type by multiplying the number of VTRACE points and the cost of this VTRACE record type. Finally we sum over all subtotals to get the total, which encapsulates most of the cost of enabled VTRACE points, namely the C_R part.

The rest of the total cost of enabled VTRACE points is independent of VTRACE record type and can be derived by multiplying the total number of intermediary enabled VTRACE points by C_T, which is platform-dependent.

- Disabled VTRACE Points

The total cost of disabled VTRACE points can be derived similarly to the C_T part of the total cost of enabled VTRACE points. The question that remains, though, is how many disabled VTRACE points there are and how they are distributed along the code. We can resolve this either by relying on the source code or by enabling all VTRACE points.

- Code-path Variation

As already discussed, code-path variation is not really an issue for lightly loaded systems; even for heavily loaded systems, our limited experience shows that VTRACE does not change the chaotic nature of the code path.

2.2.3. Verifying VTRACE Methodology

We have discussed the potential sources of perturbation VTRACE might induce and conjectured ways of correcting our results. To justify our approach, however, we need to have evidence. Here we discuss the evidence needed and how we go about collecting it.

It is obvious that to verify VTRACE methodology we need to compare what VTRACE tells us with the truth. Therefore, we need a way other than VTRACE to establish a reference frame to which we can compare the VTRACE results. The questions remaining here are how we establish the reference frame and how we make the comparisons we need.

2.2.3.1. Reference Frame

The method we choose to establish the reference frame must be much lighter in weight than VTRACE and the result it produces must be easy to compare with that of VTRACE.

2.2.3.2. Comparison

If our reference result and the VTRACE result are for the same code path under the same conditions, comparison should be easy. Otherwise, some twists are needed to find whether things match. After presenting the data in section 4, we return to verifying the methodology more concretely.

2.3. Data Collection and Analysis

2.3.1. Mechanics

The process of data collection and analysis can be broken down into the following steps:

- Specify the part of the system whose behavior is of interest.
- Write a simple test that goes through that part of the system.
- If VTRACE points have already been added to the system code, then go through the VTRACE document to find out which events are related to this part of the system.
- Start VTRACE with all those events of interest enabled.
- Start the test.
- Kill VTRACE after the test terminates and the VTRACE file is obtained.
- Define modules whose costs are of interest by bounding VTRACE points B and E (begin and end).
- Extract the costs of modules from the VTRACE file.

The module could be executed by different threads on different CPUs. However, we assume here that a single execution of the module is carried out by one thread only, which is true most of the time. To extract the cost of a module from the VTRACE file, we need to do the following:

- Locate B and E for the traced process.
- Compute the raw cost of the module by

  $$\text{RAW COST} = T_E - T_B \tag{3}$$

  where T_B is the beginning time, T_E is the ending time and we assume that throughout execution of the module no context switch has occurred.
- Clean up the VTRACE perturbation as described.

However, if a context switch does happen, we must distinguish between the following two cases:

- If a context switch happens on the CPU on which the thread executing the module is running, we are quite sure that this thread has stopped running and we should stop our timer and counter for VTRACE points correspondingly until the same thread is scheduled to run again.
- Otherwise, the thread executing the module is still running and we continue the timing.

2.3.2. Automation

The same module occurs repeatedly in the VTRACE file, and it is the average cost of the module we are interested in. It is thus most efficient to automate the above mechanics. We developed a turnkey tool with the following features:

- Module definitions can be changed easily. We use a module definition initialization file to achieve this.
- Average costs of modules are analyzed for the specified process only, i.e., the traced process.
- Interrupts and background activities are carried out in the idle process, but it is generally not easy to determine whether a certain interrupt or background event is carried out for the traced process. An approximate way to handle this is to observe such an event after the traced process has been running for a while but before it terminates. Assume that the traced process is the major player; then the hardware interrupts and background events during this period of time should belong mostly to it.
- The output from the tool gives the count of occurrences of modules, the average cost, and the standard deviation.

3. Solaris Communication Framework

Before presenting the performance data for the existing Solaris communication framework, let's briefly review the STREAMS architecture upon which the Solaris communication framework was built.

3.1. STREAMS Framework Overview

Originally developed at AT&T, STREAMS has been adopted in the Solaris implementation of UNIX. As described in the UNIX System V STREAMS Programmer's Guide, STREAMS is a collection of system calls, kernel resources, and kernel utility routines that can use and dismantle a STREAM. A STREAM is a full-duplex processing and data-transfer path between a driver in kernel space and a process in user space and has three parts: a STREAM head, module(s), and a STREAM end. The STREAM head provides the interface between the STREAM and user processes. Its principal function is to process STREAMS-related system calls. Module(s) offer a set of processing functions and associated service interfaces. The STREAM end may be a hardware driver, providing the service of an external device, or a software driver, providing functions such as loopback.

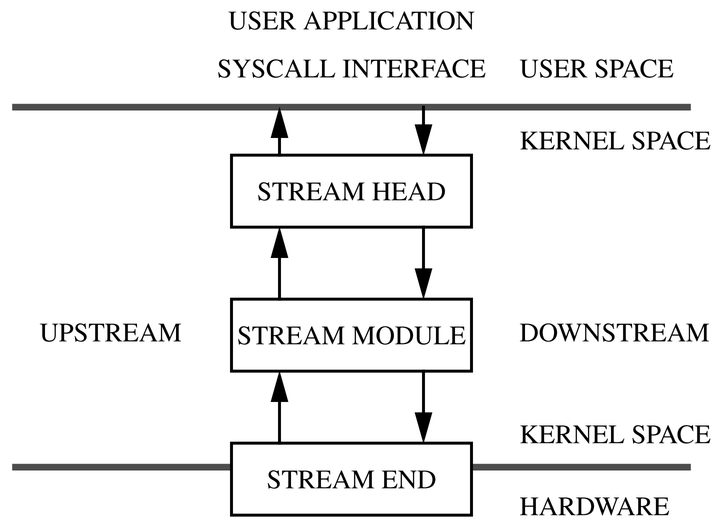

FIGURE 6. STREAM

Messages are the only means of transferring data within a STREAM. A STREAMS message contains data or status/control information or both and a specified message type. The STREAM head performs bidirectional transfer of data and information between user-space applications and kernel space STREAMS messages.

Messages traveling up or down a STREAM are processed within module(s). Each module is constructed from a pair of QUEUEs, one for upSTREAM messages, the other for downSTREAM messages. A QUEUE contains a dynamically allocated message queue, a put procedure, an optional service procedure and optional initialization and cleanup procedures, and various other control data. The QUEUE for downSTREAM messages is aware of the location of the corresponding QUEUE in the next module below it, and similarly for the QUEUE for upSTREAM messages. Modules may be dynamically added or removed from a STREAM in LIFO order, like stack operations.

STREAMS provides two flow-control mechanisms. The first is related to message-pool priority. When the STREAM head requests a message buffer in response to a *putmsg* or *write* system call, if the buffer pool becomes depleted, the STREAM head will block the user output. This applies equally to all STREAMs. The second flow-control mechanism is local to each STREAM and prevents processing bursts at any one module.

STREAMS distinguishes between ordinary messages and priority messages; only the former are subject to flow control. The general processing of messages under flow control is as follows. When flow control permits, a message is received by the put procedure in a QUEUE, which then does some or zero processing on the message and calls the STREAMS utility *putq* to put the message at the tail of the message queue. *putq* then schedules the QUEUE for execution by the scheduler in FIFO order. After some delay, the service routine is called and uses the *getq* utility to obtain the first message on the message queue. Then it calls the *canput* utility to test if the message can be forwarded to the next QUEUE. If so, it processes the message and uses *the putnext* utility to hand it to the next QUEUE. Otherwise it calls the *putbq* utility to put the message back on the head of the message queue and returns.

The STREAMS-related system calls include *getmsg*, *putmsg*, *read*, *write* and *ioctl*. It is these system calls for which we measure performance.

3.2. Solaris Communication Framework

STREAMS is the building block in the Solaris communication framework, as shown in Figure 7.

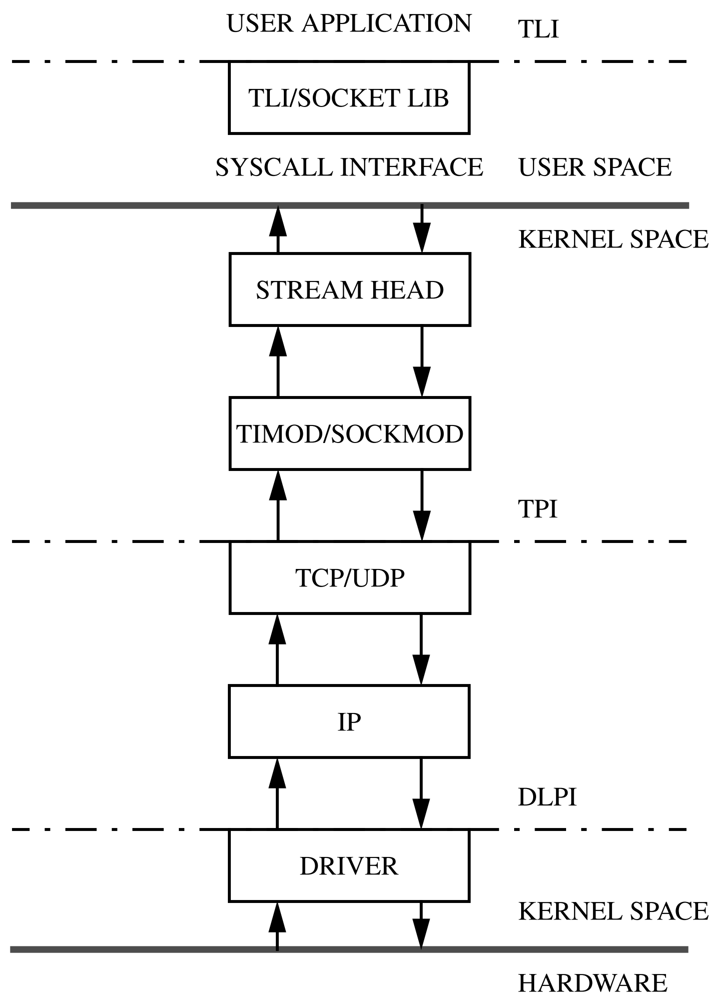

FIGURE 7. Solaris Communication Architecture

User applications access the communication facility via the transport layer interface (TLI) provided in the form of user space libraries. The transport protocols are accessed via the transport provider interface (TPI), and the drivers are accessed via the data link provider interface (DLPI). Between STREAM head and TPI are some auxiliary modules, such as TIMOD for TLI and SOCMOD for SOCKET, whose functionality is to translate STREAM messages into ones meaningful to the transport protocols.

4. Performance Analysis

We have used VTRACE as a tool to measure the performance of the various STREAMS modules that constitute an entire communication STREAM. This section discusses the test methodologies, the configuration under which tests were performed and the conventions used in data analysis, and then presents the performance data.

4.1. Test Methodology

We distinguish between tests designed for lightly loaded situations and those for heavily loaded situations, even though the latter are not thoroughly studied in this paper.

4.1.1. Light-load Case

As mentioned briefly in section 2, increasing load can produce complicated interactions inside the system and therefore provide an indicator of how chaotic the system is. Lightly loaded systems are easy to analyze but capture the essence of all STREAMS modules along the communication path. Studying lightly loaded systems provides a simple solution to performance measurement local to each STREAMS module. This itself plays an important role in performance diagnostics. For example, we can pinpoint the performance bottleneck in the entire communication path, eliminate any redundant work, and acquire a solid foundation for extending the work to general cases.

Enhancing the performance of each STREAMS module will not necessarily optimize the overall communication performance, at least not in general, but will certainly enhance it.

4.1.1.1. Ping-pong Test[1]

The ping-pong test utilizes a two-way data flow (see Figure 8) as follows. The thrower first throws a message to the reflector, which then takes all necessary steps to receive the message and reflect it back. The reflected message travels the opposite communication path from that of the thrown message and is received by the thrower, which then completes a timing cycle. Ping-pong achieves the goal of light load by a mutual synchronization mechanism, namely, the receiver reflects a message only after it receives one, and the thrower cannot put more messages into the channel until the message it sent previously has been bounced back. Hence, none of the STREAMS modules along the communication path could ever be saturated.

1. The ping-pong test was instrumented by Sun's Parallel Open Systems Group at Chelmsford, MA.

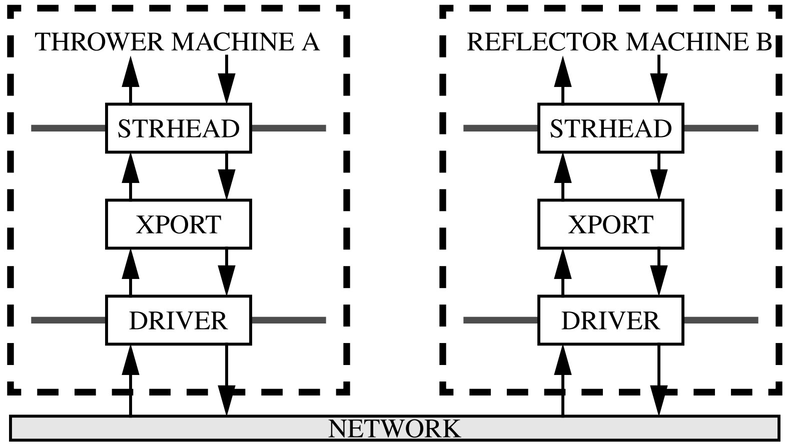

FIGURE 8. Two-way Data Flow in Ping-pong Test

The ping-pong test facilitates some additional light-weight performance measurement, such as round-trip time, using methods other than VTRACE. By doing ping-pong over either TLI/SOCKET or DLPI, one can derive the costs of intermediary STREAMS modules. For example, ping-pong implemented at the DLPI level will bypass the transport provider and deal with the data link provider directly. The round-trip time obtained at the DLPI level and that at the TPI level can yield the cost of the transport provider by simple arithmetic. This light-weight alternative method provides a good candidate for validating the VTRACE methodology.

This alternative method has drawbacks though. First, its light weight is achieved by sacrificing details. For example, its use of the ping-pong test provides no means to elucidate the code path. This could hide some interesting details, such as concurrent execution of different portions of the communication path. And the round-trip time could also include the perturbation of system background daemons that produces a distorted standard deviation from the average[1]. Secondly, it cannot be generalized to deal with situations where a significant amount of load is present.

4.1.1.2. Unidirectional Test

The unidirectional test has one-way data flow (see Figure 9). The thrower throws the message into the communication channel, which takes it all the way to the receiver. Both the thrower and the receiver can operate asynchronously at both ends of the channel at a rate appropriate to the local environment. Therefore, unlike the ping-pong test, the unidirectional test does not impose a synchronization policy. Instead we use a "regulator" to adjust the rate at which data is transmitted: the thrower executes a wait loop $\mathcal{R}(n)$ between successive sends, where n is the number of iterations of the wait loop. The parameter n is varied according to the level of regulation desired. The light-load limit corresponds to $\mathcal{R}(\infty)$, while the heavy-load limit corresponds to $\mathcal{R}(0)$.

1. To use the median instead of average is a solution to this special case.

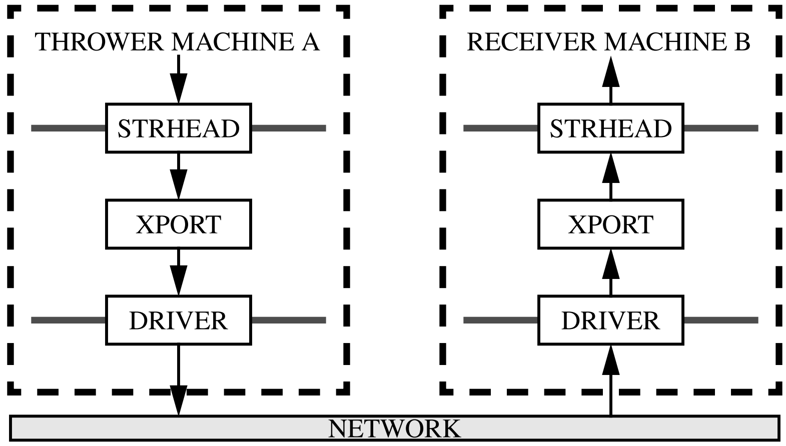

FIGURE 9. Unidirectional Data Flow

Comparing the ping-pong test with the light-load limit of the unidirectional test is not straightforward. For example, an acknowledgment is required to fulfill the requirement of reliability in a transport protocol such as TCP. The thrower must wait for the acknowledgment from the receiver after some maximum number of messages is thrown out without being acknowledged. And if the acknowledgment does not arrive after a certain amount of time, the thrower assumes that they are lost and schedules a retransmission. The receiver, after it receives the message, should acknowledge it in time to be cooperative and avoid unnecessary retransmission. Thus, acknowledgment is time-sensitive. And acknowledgment is usually represented by short messages. For two-way data flow as in the ping-pong test, acknowledgment is optimized so that the acknowledgment of data flowing one way is piggybacked on the data flowing the other way. This is impossible in a one-way data flow such as the unidirectional test. Instead, what’s been done is to put the acknowledgment message into the TCP message queue and wait for any data flowing in the opposite direction until the message times out.

The unidirectional test also allows methods other than VTRACE to measure the message traveling time. For example, the thrower could enclose the current time in the message it sends; the receiver then finds out the current time and subtracts from it the sending time enclosed in the message. This, however, omits the acknowledgment.

4.1.2. Heavy-load Case

When the load increases, so does the complexity of the code path. Load can come into play in a number of ways. The unidirectional test without regulation represents a constant load. The one-time intervention of a network-intensive event with the test represents a pulse-like load. STREAMS implements flow control to deal with load, and different transport protocols might implement their own algorithms. The interplay of these mechanisms renders the communication unpredictable on the microscopic level.

4.2. Test Configuration

Tests were performed on a stand-alone network with two dual-processor SparcStation 20s running an experimental version of Solaris at Chelmsford. During some tests one of the processors was off-line, as will be noted. The 10baseT Ethernet tests utilize the onboard LANCE Ethernet chip and the fiber channel tests utilize an experimental fiber channel driver. We assume the default configuration is used unless otherwise specified.

4.3. Conventions

Test results are presented according to the following conventions. A diagram is given for each test illustrating the details of the communication path. This is followed by a table of STREAMS module costs. The data presented here do not include the overhead of writing out the enabled VTRACE points but do contain that of testing the status for all VTRACE points, enabled or disabled (see section 2 above). As we see in the final section, the overhead is generally around 10% and depends only on the available VTRACE points along the code path. The VTRACE modules and events used for defining modules are listed in Table 1. The interrupt that frees the driver resource after a message leaves the sender machine incurs a cost different from the interrupt that takes a message to the transport after it arrives at the receiver machine. To distinguish them, we call the former free intr and the latter read intr. The experimental fiber channel driver we used in our tests were not fully traced. Instead, we measure the driver cost using the nearest TR_PUTNEXT_START and TR_PUTNEXT_END, within which the driver is enclosed. This might lead to a slightly higher driver cost (~6 μsec). In some places along the code path, there are not enough VTRACE events and yet the costs are significant. We get around the lack of VTRACE events by defining modules using existing ones. For example, the module ABOVE TCP is bounded by TR_STRWRITE_IN and TR_TCP_WPUT_IN; the module ABOVE UDP is bounded by TR_STRWRITE_IN and TR_UDP_WPUT_IN; the module STRRHEAD is bounded TR_STRREAD_ENTER and TR_STRREAD_WAKE; and the module ABOVE STRRREAD is bounded by TR_STRREAD_WAKE and TR_TCP_RPUT_OUT. The socket library was used in all the tests.

Only ping-pong test results are presented here. Results of the unidirectional test under light load are very similar.

[a]	Module Name	Bounding VTRACE Events[b]	
		Start Event	End Event
[c]	**syscall**	TR_SYSCALL_START	TR_SYSCALL_END
[d]	**intr**	TR_INTR_START	TR_INTR_END
TR_FAC_STREAMS_FR	**background**	TR_BACKGROUND_DQ	TR_BACKGROUND_DONE
	service	TR_QRUNSERVICE_START	TR_QRUNSERVICE_END
	strputmsg	TR_STRPUTMSG_IN	TR_STRPUTMSG_OUT
	strwrite	TR_STRWRITE_IN	TR_STRWRITE_OUT
	putnext	TR_PUTNEXT_START	TR_PUTNEXT_END
	strwaitq2	TR_STRWAITQ_WAIT2	TR_STRWAITQ_WAKE2
TR_FAC_BCOPY	**copyin**	TR_COPYIN_START	TR_COPY_END
	copyout	TR_COPYOUT_START	TR_COPY_END
	bcopy	TR_BCOPY_START	TR_COPY_END
TR_FAC_TCP	**tcp_rput**	TR_TCP_RPUT_IN	TR_TCP_RPUT_OUT
	tcp_wput	TR_TCP_WPUT_IN	TR_TCP_WPUT_OUT
	tcp_wsrv	TR_TCP_WSRV_IN	TR_TCP_WSRV_OUT
[e]	**udp_rput**	TR_UDP_RPUT_START	TR_UDP_RPUT_END
	udp_wput	TR_UDP_WPUT_START	TR_UDP_WPUT_END
TR_FAC_IP	**ip_rput**	TR_IP_RPUT_START	TR_IP_RPUT_END
	ip_rput_locl	TR_IP_RPUT_LOCL_START	TR_IP_RPUT_LOCL_END
	ip_wput	TR_IP_WPUT_START	TR_IP_WPUT_END
	ip_wput_ire	TR_IP_WPUT_IRE_START	TR_IP_WPUT_IRE_END
TR_FAC_LE	**le_intr**	TR_LE_INTR_START	TR_LE_INTR_END
	le_wput	TR_LE_WPUT_START	TR_LE_WPUT_END
	le_freebuf	TR_LE_FREEBUF_START	TR_LE_FREEBUF_END

TABLE 1. Module Definitions

a. VTRACE Facility Name
b. See /usr/include/sys/vtrace.h for details.
c. TR_FAC_SYSCALL
d. TR_FAC_INTR
e. TR_FAC_UDP

4.4. Ping-pong Tests Over Fiber Channel

Ping-pong tests were performed over TCP and UDP for a number of message sizes.

4.4.1. TCP

Diagrams of *write*, *read* system calls and related interrupts (Figures 10-12) are followed by performance data (Table 2). The left-hand side of a diagram shows the flow of control and the right-hand side shows the modules and the VTRACE points we used to bound them.

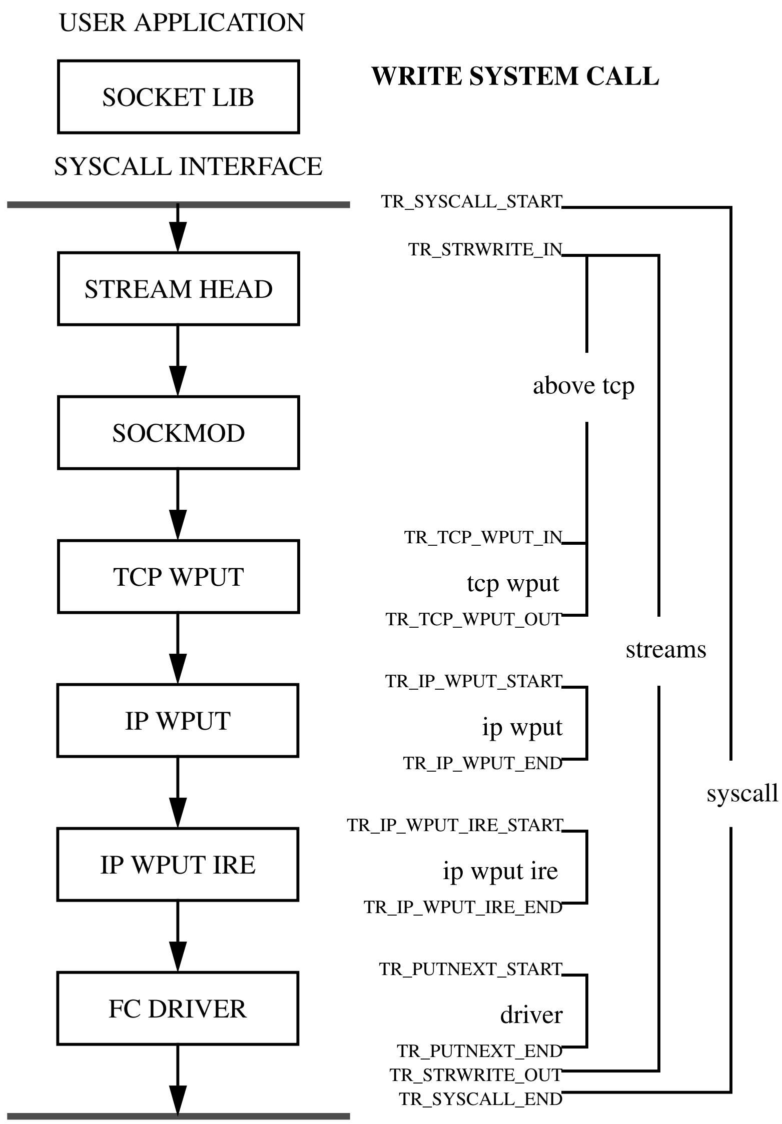

FIGURE 10. Write System Call

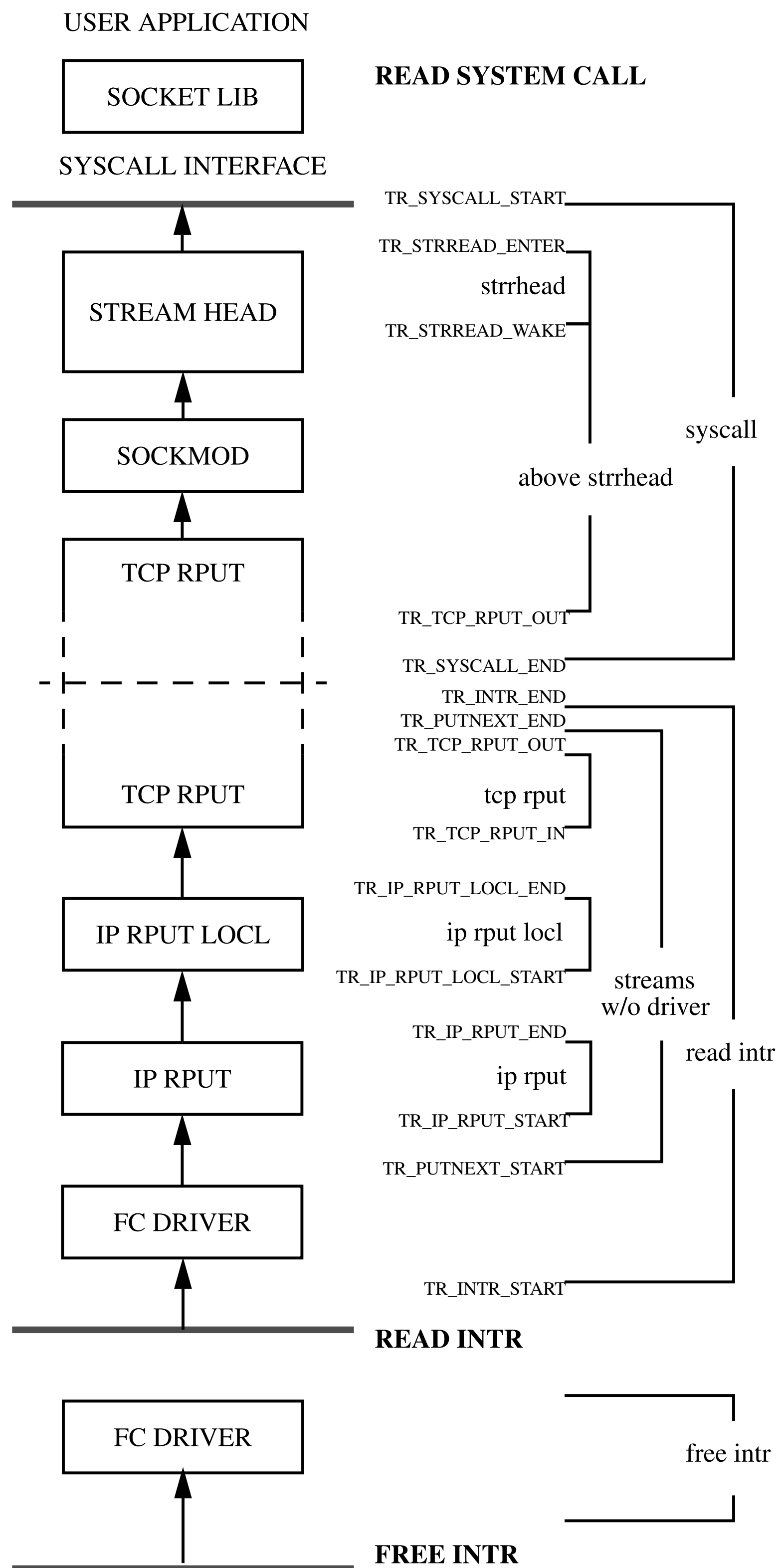

FIGURE 11. Read System Call and Interrupts (Large Messages)

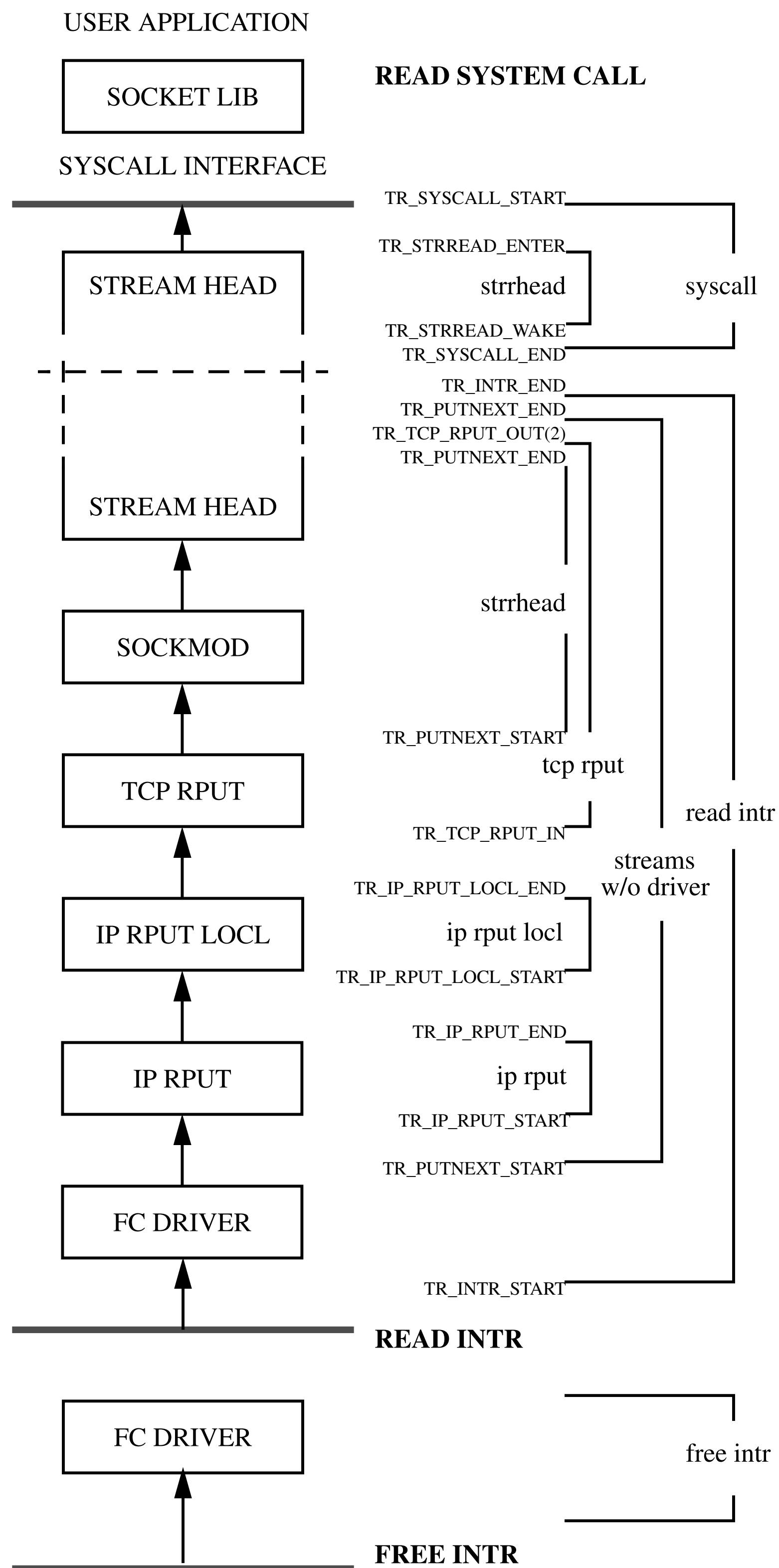

FIGURE 12. Read System Call and Interrupts (Small Messages)

	Module Name	Message Sizes (Bytes) Time (µsec) and Standard Deviation (µsec)					
		32		1400		4096	
		Time	**Std**	**Time**	**Std**	**Time**	**Std**
WRITE SYSCALL	**WRITE SYSCALL**	171	2.9	212	2.2	218	8.5
	SETUP	18	-	20	-	19	-
	STREAMS	153	3.1	192	2.1	199	7.6
	ABOVE TCP	37	0.9	67	1.0	76	4
	TCP WPUT	23	0.6	24	0.7	23	1.2
	IP (WPUT + IRE)	17	1.9	17	0.8	19	4.8
	DRIVER	53	1.9	58	1.1	58	1.1
	OTHER	23	-	26	-	23	-
READ SYSCALL	**READ SYSCALL**	70	2.7	177	2.6	221	3.5
	STRRHEAD	36	2.0	44	2.3	38	1.1
	STRWAITQ2	25	1.9	30	2.1	26	1.0
	ABOVE STRRHEAD	-	-	93	1.7	143	2.5
	OTHER	34	-	40	-	40	-
FREE INTR	**FREE INTR**	42	0.8	39	0.8	39	0.7
	SETUP	-	-	-	-	-	-
	DRIVER	-	-	-	-	--	-
	FREE RESOURCE	2	1.0	2	1.1	2	0.7
	OTHER	-	-	-	-	-	-
READ INTR	**READ INTR**	158	3.6	128	1.6	131	1.8
	SETUP + DRIVER	62	-	68	-	71	-
	STREAMS W/O DRIVER	96	1.8	60	1.2	60	1.8
	IP (RPUT + LOCL)	8	0.8	9	1.3	8	1.0
	TCP RPUT	73	1.3	41	0.7	40	1.8
	STRRHEAD	37	1.0	-	-	-	-
	OTHER	34	-	-	-	-	-
	OTHER	15	-	10	-	12	-

TABLE 2. Ping-pong Tests of TCP over Fiber Channel

Module names are indented in Table 2 to show the caller/callee relationship between modules as if they were functions. For example, in the *write* system call, SYSCALL consists of SETUP and STREAMS, and STREAMS itself calls a number of submodules, such as ABOVE TCP, TCP WPUT, IP, DRIVER, and OTHER. From the measurement point of view, the pair of VTRACE points that defines a module is fully and immediately (without going through any intermediary modules) contained in the pair that defines its parent (see Figures 10-12 for details). The cost of a STREAM module is the sum of the total costs of its children.

Standard deviations are supplied for all measured costs. Some costs are simply placeholders for the unmeasured portions in the corresponding parent modules. These costs do not have accompanying standard deviation fields since they are derived by subtracting from the parent module cost all measured sibling module costs.

In the *read* system call, the definitions of modules are somewhat obscure, e.g., we do not know exactly where the STRRHEAD ends. This could be improved by adding more VTRACE points into the source code.

The IP cost is the sum of two measured module costs, IP_WPUT and IP_WPUT_IRE. The standard deviation is obtained by summing over those of the components.

The shaded rows exhibit a clear dependency on message size: the modules that the shaded rows represent are very probably doing things like copying and check-summing that involve much data movement.

It is interesting to see that the IP cost is affected very little by varying message size, even though it is supposed to perform check-summing, perhaps because the check-summing is done while copying the message. Also, transfer of the message from kernel space to driver apparently does not require copying, since the driver cost is almost invariant with message size.

Notice that transferring small messages has been optimized. Large messages, e.g., the 1400-byte and 4096-byte messages, are handed from the driver to TCP in the interrupt context and stay there until they are picked up. And it is the *read* system call that does the copying and check-summing to get the message out of TCP to the STREAM head and then to the user. However, smaller messages, like the 32-byte message, are passed from the driver, through TCP, directly into the STREAM head in the interrupt context, which saves the *read* system call from doing anything else but taking the message out of the STREAM head to user space. This does increase the weight of the interrupt by a certain amount, but makes the read system call extremely cost-efficient. As we will see for UDP, the message, no matter of what size, is always transferred from the driver, through UDP, to the STREAM head in the interrupt context. Therefore, transferring small messages over TCP is optimized to have at least the UDP performance while reliability is retained.

In the measurement, we collected 81 samples for system calls and related modules, 41 for *free intr* and 40 for *read intr*.

4.4.2. UDP

The diagrams of *putmsg*, *getmsg* system calls and related interrupts (Figures 13 and 14) are followed by performance data (Table 3). The left-hand side of a diagram shows the communication path and the right-hand side shows the VTRACE points we used to bound modules.

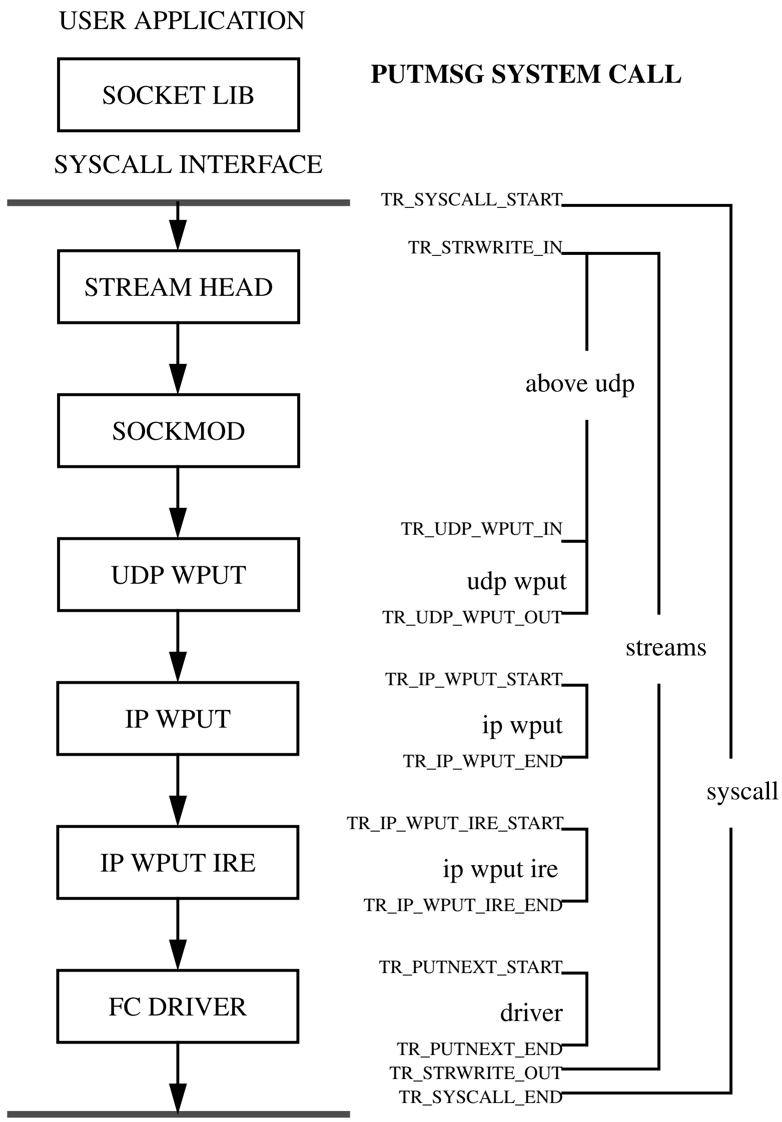

FIGURE 13. Putmsg System Call

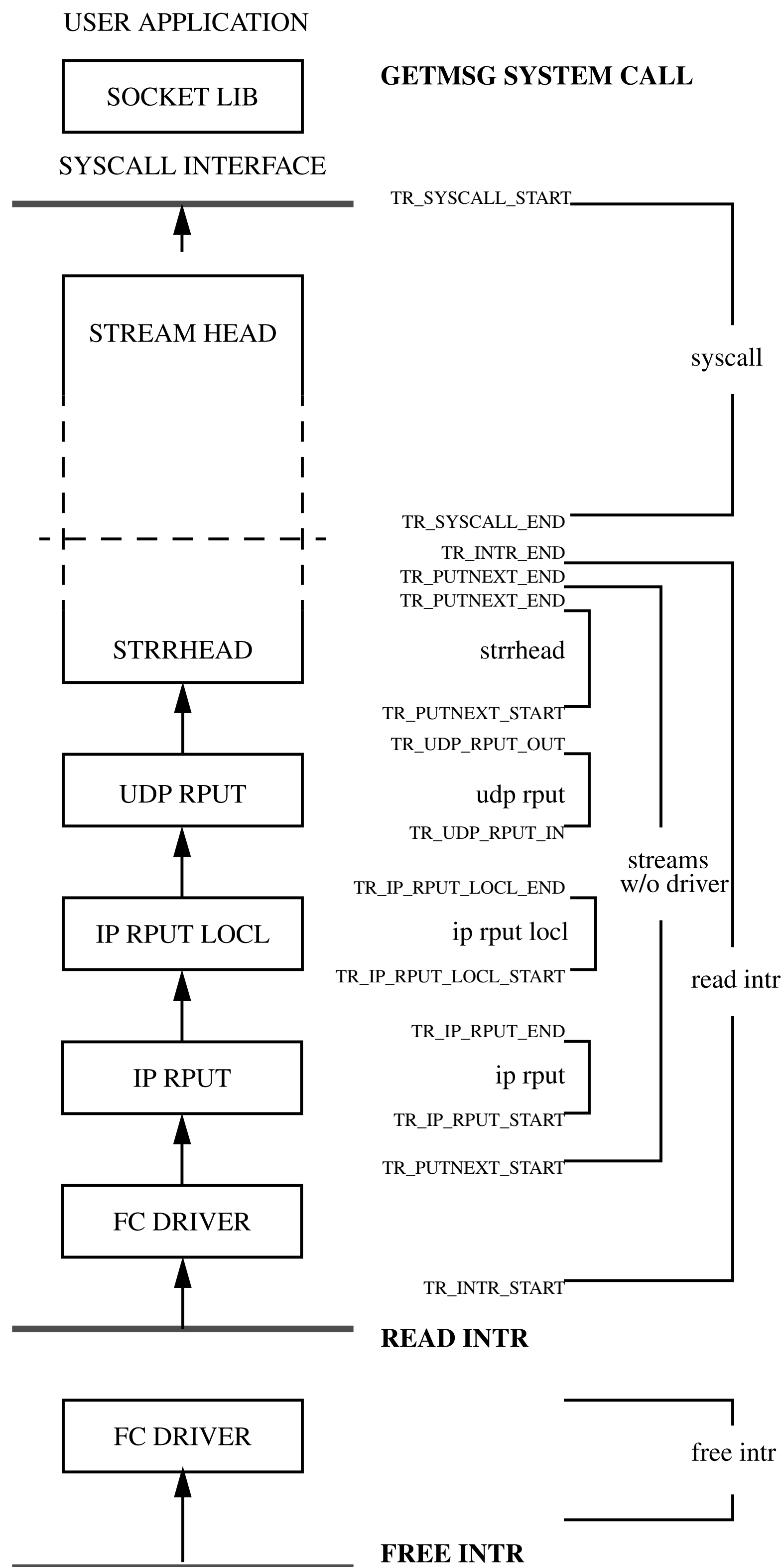

FIGURE 14. Getmsg System Call and Interrupts

	Module Name	32 Time	32 Std	1400 Time	1400 Std	4096 Time	4096 Std
		Message Sizes (Bytes) Time (μsec) and Standard Deviation (μsec)					
PUTMSG SYSCALL	**PUTMSG SYSCALL**	174	3.2	244	3.3	312	8.7
	SETUP	25	-	25	-	37	-
	STREAMS	149	2.7	219	3.3	275	8.4
	ABOVE UDP	52	2.1	91	1.8	126	6.2
	UDP WPUT	5	0.5	5	0.5	6	0.4
	IP (WPUT + IRE)	21	0.8	38	2.0	50	2.7
	DRIVER	54	0.9	65	2.0	73	2.6
	OTHER	17	-	20	-	20	-
GETMSG	**GETMSG SYSCALL**	88	3.8	119	2.3	190	6.0
	STRWAITQ2	22	0.7	27	0.8	32	2.4
	COPYOUT (5)	7	-	30	-	61	-
	OTHER	59	-	62	-	97	-
FREE INTR	**FREE INTR**	47	1.4	44	2.8	42	0.7
	SETUP	-	-	-	-	-	-
	DRIVER	-	-	-	-	-	-
	FREE RESOURCE	2	0.8	3	1.2	3	1.0
	OTHER	-	-	-	-	-	-
READ INTR	**READ INTR**	128	2.4	171	3.2	220	4.0
	SETUP + DRIVER	58	-	68	-	68	-
	STREAMS W/O DRIVER	70	2.4	103	2.5	152	3.7
	IP (RPUT + LOCL)	16	1.0	47	2.5	87	2.7
	UDP RPUT	8	0.7	9	0.4	10	0.5
	STRRHEAD	34	2.2	36	1.1	41	2.3
	OTHER	12	-	11	-	14	-

TABLE 3. Ping-pong Tests of UDP over Fiber Channel

In the results reported in Table 3, COPYOUT is called five times in the *read* system call; the cost given here is the total cost.

The IP cost increases with the message size since check-summing is byte-intensive.

For the test using 32-byte messages, we collected 71 samples for system calls and related modules, 36 for *free intr* and 35 for *read intr*. For tests using larger messages, we collected 81 samples for system calls and related modules, 41 for *free intr* and 40 for *read intr*.

4.4.3. TCP v.s. UDP

Module Name	Message Sizes (Bytes) TCP Time (µsec) and UDP Time (µsec)					
	32		1400		4096	
	TCP	**UDP**	**TCP**	**UDP**	**TCP**	**UDP**
WRITE/PUTMSG SYSCALL	171	174	212	244	218	312
READ/GETMSG SYSCALL	70	88	177	119	221	190
FREE INTR	42	47	39	44	39	42
READ INTR	158	128	128	171	131	220

TABLE 4. Ping-pong Tests Over Fiber Channel - TCP versus UDP

Table 4 shows the corresponding costs using TCP and UDP. It is clear that TCP outperforms UDP for large messages and also approaches UDP in the small message limit.

IP check-summing has been highly optimized in TCP but not in UDP.

The ABOVE UDP cost is much higher than the ABOVE TCP cost for the same message size.

Since UDP does not have reliability control, it puts all messages, regardless of size, into the STREAM head after they arrive. TCP only does this for small messages; it leaves larger ones in its own message queue to be picked up.

In UDP, the STREAM head might just throw the arriving messages away if the system-wide buffer pool is overused. However, how TCP achieves reliability after it passes the small messages to STREAM head would be interesting to know.

5. Verifying VTRACE Methodology

Now that we have presented the performance data for various test methodologies, transport protocols and drivers, we return to verifying the correctness of the VTRACE methodology using the data at hand.

As mentioned in previous sections, with the ping-pong test we can use a lightweight alternative method to verify the VTRACE methodology. In this section, we discuss in detail the correctness criteria for the VTRACE results and how we go about testing them.

5.1. Criteria

The VTRACE methodology we have presented should meet at least the following criteria:

1. Both the code path and the various module costs should be reproducible.
2. The code path should not depend on what or how many VTRACE points we choose to enable.
3. After the overhead for writing out all enabled VTRACE points is removed, the VTRACE results should be invariant to the number of VTRACE points we choose to enable.
4. After all sources of perturbations induced by VTRACE are removed, the results should match the results obtained without the influence of VTRACE in the first place.

The criteria overlap but do, we believe, reflect the truth from different points of view.

5.2. Evidence

We demonstrate the validity of the VTRACE methodology by the following two examples.

First, we performed VTRACE on the same test twice but with different sets of VTRACE points enabled: we enabled only a subset of VTRACE points during the second run. Of course the raw costs, i.e., the costs including the VTRACE overhead, should be different; however, the costs with the overhead for writing out all enabled VTRACE points taken out should match up.

During the first run, we enabled all VTRACE points in the following facilities:

```
vtrace -f syscall intr streams_fr tcp ip le kmem bcopy
```

During the second run, we selectively enabled the following VTRACE points in the above facilities:

```
vtrace -f syscall intr tcp -e ip ip_wput_start
       ip_wput_end ip_wput_ire_start ip_wput_ire_end
       ip_rput_start ip_rput_end ip_rput_locl_start
       ip_rput_locl_end -e le le_wput_start le_wput_end
       le_intr_start le_intr_end le_read_start
       le_read_end -e streams_fr strwrite_in
       strwrite_out
```

The data showing the results before and after the cleanup are given in Table 5. The results are exactly as expected.

	Module Name	Tests[a]	
		First Run	Second Run
BEFORE CLEANUP	**WRITE SYSCALL**	393	461
	STREAMS	345	414
	TCP WPUT	47	51
	IP (WPUT + IRE)	43	62
	DRIVER	89	114
	OTHER	156	187
AFTER CLEANUP	**WRITE SYSCALL**	363	368
	STREAMS	321	325
	TCP WPUT	44	43
	IP (WPUT + IRE)	38	43
	DRIVER	87	89
	OTHER	152	150

TABLE 5. Rectifying VTRACE Perturbation from Writing VTRACE Records

a. These tests were performed on a different platform from that used in section 4.

Second, we performed two runs of the ping-pong tests. In the first run, none of the VTRACE points were enabled and we used the alternative ping-pong method to obtain the round-trip time. In the second run, we used VTRACE instead and obtained a table of module costs similar to those presented in section 4; everything else was kept the same. The bottom line is to compare the results and see whether they match or not.

We cannot, however, add together all the component costs measured from VTRACE and expect the sum to be the same as the round-trip cost given by ping-pong measurement, since some of the modules may not be accounted for in the ping-pong round-trip measurement and some modules may be executed concurrently so as to be counted twice by the simple-minded summation. So let's first examine the code path in the time chart in Figure 15.

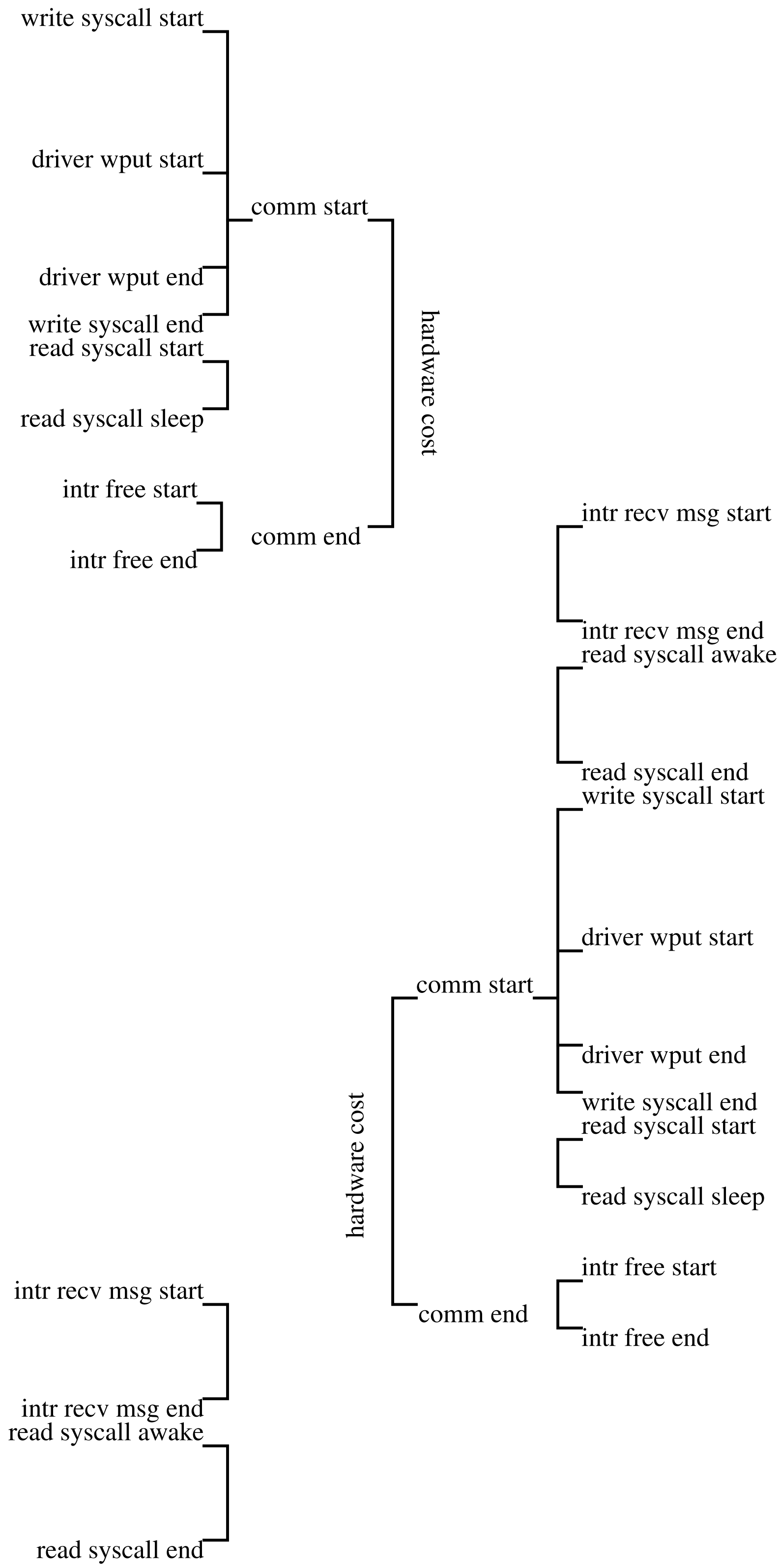

FIGURE 15. Ping-pong Test Time Chart

This time chart shows that the data gets to the hardware before the write syscall terminates and that part of the read system call and part of the *free intr* overlap with the hardware cost. The amount of overlapping is test-specific.

The second run is the same ping-pong test we used to obtain the 32-byte message UDP result. And the first run yields a round-trip time of 1068 μsec.

Before we can make the comparison, we must also find out the hardware cost, which can be calculated by the following formula:[1]

H/W COST = H/W_LATENCY + MESSAGE_SIZE / H/W_BANDWITH (4)

The hardware latency is measured at 180 μsec from the time when software places a message directed to the second adapter port on the adapter output queues to the time when the interrupt dispatch occurs. The fiber channel hardware bandwidth is 25MB/sec. Therefore, the hardware cost for a 32-byte message is calculated to be 181 μsec.

Adding approximately half the driver cost, the rest of the *write* system call cost, the *read intr* cost and the second part of the read system call cost gives us a total of 543 μsec, which, when doubled, is pretty close to the yield from the first run. The arithmetic is shown below.

	PUTMSG	174
	GETMSG	88
	FREE INTR	47
	READ INTR	128
+	HARDWARE COST	181
	SUBTOTAL	618
	HALF OF THE DRIVER COST	27
	READ START TO SLEEP	8
-	OVERLAPPING FREE INTR	40
	TOTAL	543

Finally we show in Table 6 the VTRACE perturbation arising from testing the status of a VTRACE point. We see that VTRACE increases the cost for all message sizes by almost a constant amount, which is dependent only on the code path. The ratio of the perturbation to the total cost is around 1/10.

1. The hardware cost and related work was done by Sun's Parallel Open Systems Group at Chelmsford, MA.

Message Sizes (Byte)	Test Condition (All Times in µsec)					
	TCP			UDP		
	NO VTRACE	VTRACE OFF	DIFF %	NO VTRACE	VTRACE OFF	DIFF %
32	411	465	13.1	490	535	9.2
1400	596	644	8.1	687	745	8.4
4096	816	886	8.6	984	1025	4.2

TABLE 6. VTRACE Perturbation Common for All VTRACE Points

6. Conclusion

In this paper we applied VTRACE methodology to communication performance measurement. In particular, we measured the performance of a couple of light-load tests using both TCP and UDP as the underlying transports.

We analyzed the code paths and collected costs of the component modules. Unlike most lightweight measurement techniques, VTRACE made it possible to follow the footprint of a program's execution, which we found extremely useful on understanding the complicated algorithms used to implement protocols and some dynamic behaviors such as that of the STREAMS under flow control.

Although the information provided by VTRACE is extremely useful, an unsparing use could lead to overhead costs well over 100%. Because the overhead of VTRACE is spread all over the code path, it seems hard to characterize to what extent the target is perturbed. In this paper, we examined in fine detail the impact of VTRACE itself on our results and explored ways to minimize it. We argued that the perturbation could be categorized into two kinds: local perturbation and global perturbation. By using methods discussed in Section 2, we were able to reduce the former down to less than 10%. We also argued that the latter did not affect our results, but could have arised if the load were heavy.

The cost of data manipulation, such as copying and checksumming, is generally message-size dependent. Using this as an indicator, we were able to locate a number of modules that manipulate data in certain ways. Such modules include ABOVE TCP/UDP, IP, and DRIVER. This suggests that data are copied at least twice: from user space to kernel space, and from kernel space to network board. Because data manipulation is costly, elimination of unnecessary copies, combination of checksumming and copy, and similar techniques can reduce the communication overhead by a fair amount. Such techniques were used in the test system's TCP implementation, which explains why the cost for transmitting a message using TCP is smaller than that of UDP even though TCP incurs more protocol processing overhead. The rest of the cost was found to spread over the entire stream, which indicates the overhead of the STREAMS framework is not negligible.

In general, we saw that VTRACE provides a solid framework for performance work. In the future we would like to apply the VTRACE methodology to more complicated situations, such as a series of heavy-load cases. We will examine to what extent, if any, our results are subject to the more complicated global perturbation, and how to eliminate it. As the load is increased, so will be the irregularity of the code path.

Moreover, the costs of certain modules will become more and more "context sensitive" (they are dependent upon the state of the entire stream, such as the various STREAMS queues). The simple summary statistics will be rendered far less effective as it is for the light-load cases. So new techniques have to be devised to present the data in an efficient and easy-to-understand way.

7. Acknowlegements

We are grateful to Barry Medoff and Sun's Parallel Open Systems Group at Chelmsford, MA for their advice, criticism, and direction for this work. Special thanks to Katrina Avery for copyediting.

8. References

[1] STREAMS Programmer's Guide, *AT&T*

[2] Internal Documentation on VTRACE, *Sun Microsystems, Inc.*