A Stall Metric to Track Communication Performance

Alan Mink, Wayne Salamon and Michael Indovina

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{amink@nist.gov}

Abstract---

Probing the communication protocol stack in Linux PC-based clusters to investigate erratic TCP/IP performance has led to a new metric, data stream stall, which is analogous to instruction stream stall in CPUs. Data stream stalling correlates well with unexpected throughput performance dips; the dips are usually due to delayed ACKs or questionable handling of them. We illustrate the use of this data stream stall metric by isolating and correcting the cause of these communication throughput dips in our version of Linux (2.0.29). The availability of this data stream stall metric would provide useful feedback to users by indicating deficient communications performance.

Keywords--- **ATM, Communication Protocols, Fast Ethernet, Linux, Performance Measurement , TCP/IP.**

I. INTRODUCTION

While evaluating the performance of MPI based applications [IND98] on commodity clusters [BEC95], using both ATM and Fast Ethernet communications, we encountered a communications performance anomaly. This anomaly manifested itself as a sharp degradation in TCP/IP communications throughput for slightly different message sizes. To probe the cause of this anomaly, we used the NIST developed low perturbation MultiKron® performance data collection instrumentation and NIST time synchronization instrumentation to delve into the Linux communication protocol. Using the time synchronized MultiKron we were able to measure directly the communication latency between nodes and across switches, rather than infer latency from average round trip times. We were also able to obtain accurate measurements of processing rates and latencies within the various communication protocol layers.

In probing the Linux communication protocol stack to investigate the erratic TCP/IP performance we developed a new metric, data stream stall. This metric closely tracks the observed unexpected throughput performance. Using this metric along with MultiKron performance probes we were able to isolate the cause of the throughput dips and correct them in our version of Linux (2.0.29).

II. MEASUREMENT INSTRUMENTATION

Current NIST instrumentation consists of the MultiKron_II [MIN94] custom VSLI chip and its associated toolkits [MIN95,MIN97] for standard I/O buses (currently VME, SBus, and PCI). The MultiKron_II provides a high-precision clock, event tracing and 16 performance counters. Performance counters can be used to count the number of occurrences of a target event or as a stopwatch to record the elapsed time between events.

Operationally the MultiKron is a passive, memory-mapped device. Programmers interact with MultiKron via reads and writes to the mapped memory region. To generate traces, a software measurement event is triggered by the execution of a specific statement, a measurement probe, inserted into the application program. A measurement probe can be added to the source code, requiring recompilation, or added directly to the executable code via a binary patch [HOL97].

The probe instruction appears as an assignment statement to a memory mapped MultiKron address. When the measurement probe code is executed, the data value from the assignment statement (generally an event ID) is written to the MultiKron. The MultiKron then appends its current timestamp (precision on the order of 100 ns), the identity of the process, and (in multi-processor systems) the identity of the CPU, to that data to form an event trace sample. The trace sample is then automatically buffered and written to the MultiKron data storage interface. Operating system support is necessary during context switches to load the MultiKron with the current process ID. Thus, MultiKron provides a hardware assist to traditional software-based instrumentation systems thereby minimizing the perturbation to the executing program by simplifying the probe code to a single write operation and recording the time-stamped samples into the MultiKron's own memory.

Current software techniques [LEV95,MIL92] can attain time synchronization between cooperating computers on the order of 1 ms. This accuracy of time synchronization is

insufficient for system level performance measurement in cluster computing environments, where time intervals are in the range of tens to hundreds of microseconds. NIST has developed time synchronization instrumentation that integrates with the MultiKron toolkits. A version of this instrumentation, which operates in a local cluster environment, supplies MultiKron devices on all the nodes of a cluster with a common clock and a common reset signal, so that all MultiKron clocks will advance in lock-step. Thus the time synchronization achieved is better than 25 ns. Another version of this instrumentation can operate in a global environment by using pulses from the global positioning system (GPS) to "train" a local oscillator. These pulses occur once per second to within an accuracy of 1 μs. The global version will supply geographically distributed MultiKrons with a clock and a reset signal, in the same way as the local version, but not in lock step -- only to within 1 μs accuracy. Both versions of this time synchronization instrumentation are operational, but the local version was used in these measurements.

III. ATM AND FAST ETHERNET PERFORMANCE

The anomalous communication behavior that we encountered is illustrated by the performance of our micro-kernel, shown in Figure 1. This is a standard plot of the communication throughput versus the message size for both ATM and Fast Ethernet in which we see large dips that degrade throughput by as much as 80% for some message sizes. In contrast, once the problem was found and corrected we obtained the curves of Figure 2. Figure 1 contains two curves, one for ATM and one for Fast Ethernet, while Figure 2 contains four curves, two for ATM and two for Fast Ethernet. The two ATM and two Fast Ethernet network interface cards (NIC) are all from different manufacturers. The two Fast Ethernet curves are so close they appear as one curve.

To determine the cause of this anomalous behavior we inserted MultiKron probes into the application, socket, TCP/IP and device driver layers of the kernel communications protocol on both the source and destination nodes. Although this provided us with selective event timing interval information, it didn't provide us with any overall feedback as to when performance was being degraded. This desire for an overall indicator was the basis of the stall metric. We wanted to know when the communication stack was waiting, or stalled. What we needed was a "stopwatch" which would accumulate the stall time by re-starting when a protocol stack stall occurred and by pausing when the stack was active or complete. We used one of the MultiKron performance counters, which is capable of functioning as a stopwatch, to accumulate the stall time. Alternatively, we could have built software routines around the Pentium clock to obtain the abstraction of a stopwatch, but instead we opted for the more direct, lower perturbation MultiKron approach. We define stall time as any period when the device driver is not active and the protocol software waits for an event to occur, either by sleeping or enqueueing packets. We consider the device driver active when its software is executing or the NIC is active. When the protocol software waits for an event to occur and the device driver is not active, the stopwatch is re-started. When the protocol software or the device driver becomes active, the stopwatch is paused. When the protocol software completes the stopwatch is also paused. At any time during or after program execution we can read the stopwatch to determine whether a communication stall has occurred along with the associated performance degradation.

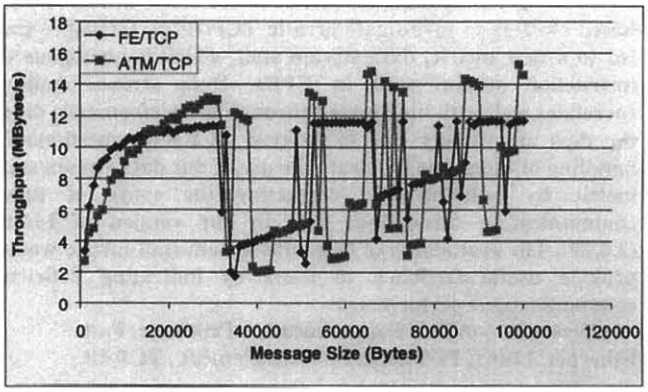

Fig. 1 Fast Ethernet and ATM Over TCP/IP Throughput Original.

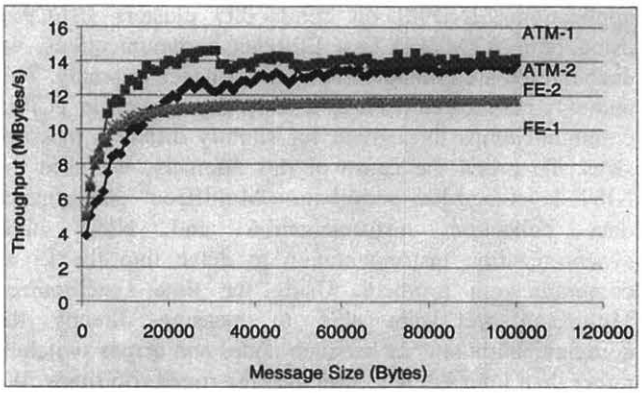

Fig. 2 Fast Ethernet and ATM Over TCP/IP Throughput Corrected.

A discussion of our stall algorithm follows. Upon entering the protocol software, pause the stopwatch. At any stall point, before stalling, if the device driver status is not active re-start the stopwatch. When the stall is completed, pause the stopwatch before continuing with the protocol software. When entering the device driver, set its state as active and pause the stopwatch. When the NIC interrupts

with a transmit complete status signal, if any stall events are occurring then re-start the stopwatch. Pausing a paused stopwatch or re-starting a counting stopwatch has no effect. In the Linux 2.0.29 kernel there are three stall points in the TCP/IP protocol software. One stall point is when a packet buffer is allocated which will be used to copy the segmented user data into the kernel. If there is not enough available memory to complete the segmentation, the protocol software will sleep, waiting to be awakened when memory becomes available. The other two stall points are when the transmission window is full or a partial packet is to be sent, causing these packets to be placed on the transmission queue, waiting for incoming ACKs to open up the window. One complication in obtaining the stall metric is that it requires information from independently operating protocol layers – TCP/IP and the device driver/NIC. The TCP/IP software operates in parallel with the NIC, which is a separate hardware device managed by a device driver. Thus, it becomes necessary to implement a set of *flags*, which are accessible between the two software layers asynchronously. These *flags* are implemented in the socket data structure, thus providing a per-socket metric.

Through a combination of the timed event traces and the stall metric we were able to determine the cause of the anomalous behavior. It seems that the protocol implementation was erroneously handling the transmission of partial packets, although the slow start algorithm did contribute slightly. Our 2.0.29 Linux implementation provides for two options in the handling of partial packets. One option is trying to conserve network resources by delaying the transmission of partial packets, waiting for a timeout or additional message data to become available within a short time so a full packet can be sent rather than just a partial packet. This option is the default. The other option is to send partial packets as soon as possible. This option can be invoked via "TCP_NODELAY" parameter. Due to a bug in this code[1], when a partial packet was delayed because of insufficient space in the transmission window, instead of it being sent immediately when space became available it was erroneously being delayed until the timeout occurred. The stall metric is plotted in Figure 3 and tracks well with the observed performance dips. The stall metric has been normalized for display here. The performance of the corrected version of TCP/IP, shown in Figure 2, displays none of the anomalous dips in performance that occurred in the original version.

The slow start algorithm and the last partial segment delay are mechanisms that optimize network traffic (a system metric) rather than user throughput (a response metric). These two classes of metrics are normally inversely related, so that enhancing one degrades the other. In switched LAN environments, which are common for most cluster computing configurations, there should be a means to over-ride these mechanisms so that specific applications can realize their full potential of these fast network technologies. The "TCP_NODELAY" parameter provides the means to disable partial packet delays, but there is no way to disable the slow start algorithm.

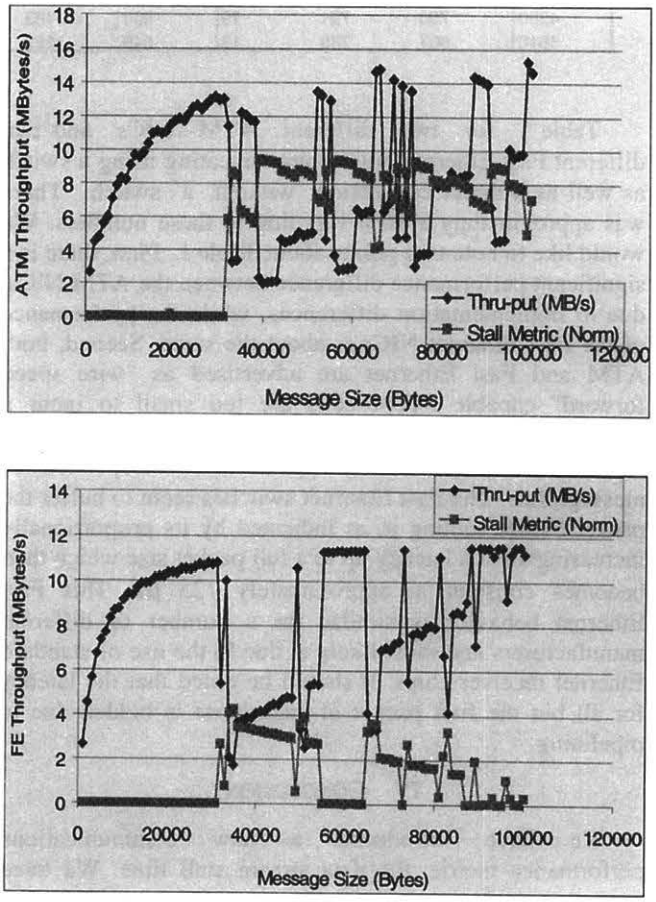

Fig. 3 ATM and Fast Ethernet Throughput, Respectively, Plotted Along with its Associated Normalized Stall Metric.

During the course of our measurements we obtained the application layer send-receive latencies of both ATM and Fast Ethernet messages. A MultiKron timestamp was acquired on the sending machine immediately prior to the "send" and another synchronized timestamp was acquired on the receiving machine immediately following the "receive". This provided a direct measurement of latency versus one inferred from round-trip time. The results of these send-receive latency measurements are shown in

[1] For a complete description of the bug in 2.0.29 refer to *www.cmr.ncsl.nist.gov/scalable/misc_info/Linux_TCP.html*

TABLE I

APPLICATION LAYER "SEND-RECEIVE" LATENCIES, IN µS, OF TCP/IP OVER ATM AND FAST ETHERNET. WE ARE USING TWO DIFFERENT INTERFACE CARDS FOR NETWORK TECHNOLOGY AND TWO CONNECTION METHODS FOR EACH CARD, SWITCH (A SWITCH CONNECTING THE NODES) AND DIRECT (NO SWITCH, JUST A WIRE CONNECTING THE TWO NODES).

Send	ATM-1			ATM-2			FE-1			FE-2		
Bytes	Switched	Direct	Sw Delay	Switched	Direct	Sw Delay	Switched	Direct	Sw Delay	Switched	Direct	Sw Delay
4	127	111	16	143	126	17	89	81	8	83	78	5
1400	318	302	16	243	225	18	366	235	131	346	228	118
1460	326	309	17	250	228	22	369	245	124	353	226	127
2920	525	504	21	364	346	18	491	364	127	474	349	125
4380	750	731	19	500	483	17	614	491	123	596	472	124
5840	807	789	18	549	527	22	743	612	131	720	593	127

Table I for two different ATM NICs and two different Fast Ethernet NICs communicating using a switch as well as a direct connection without a switch. There was approximately a 10% variation in these numbers. We would like to note two points about Table I. First, there is a significant performance difference between the ATM NICs, due to implementation differences, while the performance of the Fast Ethernet NICs is about the same. Second, both ATM and Fast Ethernet are advertised as "wire speed forward" capable. ATM cells are too small to incur a significant overhead due to buffering. The approximately 20 µs ATM switch latency appears to be independent of message size. The Fast Ethernet switches seem to buffer the packet before routing it, as indicated by its proportionally increasing switch latency up to a full packet size which then becomes constant at approximately 125 µs. This Fast Ethernet behavior is similar for a number of different manufacturers and most likely is due to the use of standard Ethernet receiver chips. It should be noted that the latency for all but the first packet of each burst is hidden due to pipelining.

IV. CONCLUSION

We have introduced a new communications performance metric, the data stream stall time. We have used this metric to identify communication performance degradation in the TCP/IP protocol stack of the Linux (2.0.29) kernel. After determining and correcting the cause of this performance degradation, which was due to erroneous handling of partial packets, we demonstrated significantly improved communication performance. In addition, we have shown a performance differential between two different ATM NICs, which contrasts with the uniform performance between two different Fast Ethernet NICs. We have also shown that many Fast Ethernet switches buffer, rather than cut-thru route, incoming packets before routing.

Using an open operating system, such as Linux, yields a significant benefit by providing source code access. Source code access allows one to understand the specific implementation details, to instrument the code for performance measurement, and to make modifications to the kernel to obtain customized performance and/or functionality.

REFERENCES

[BEC 95] D. Becker, T. Sterling, D. Savarse, U. Ranawake and C. Packer, *BEOWULF: A Parallel Workstation for Scientific Computation*, Proc. of the International Conf. on Parallel Processing, Urbana-Champaign, IL, Vol. I: Architecture, pp I11-I14, Aug. 1995.

[HOL 97] J. K. Hollingsworth and B. Buck, *DyninstAPI Programmer's Guide*, CS-TR-3821, University of Maryland, Aug. 1997.

[IND 98] M. Indovina, A. Mink, R. Snelick and W. Salamon, *Performance Measurement of ATM and Ethernet Computing Clusters*, Proc. of ATM98 Developments Conf., Rennes, France, Vol. II, pp 43-64, Mar. 1998.

[LEV 95] J. Levine, *An Algorithm to Synchronize the Time of a Computer to Universal Time*, IEEE Trans. on Networking, Vol. 3, No. 1, pp 42-50, Feb. 1995.

[MIL 92] D. Mills, *Network time protocol (version 3): specification, implementation and analysis*, DARPA Network Working Group Rpt. RFC-1305, Univ. of Delaware, 1992.

[MIN 94] A. Mink, *Operating Principles of the MultiKron_II Performance Instrumentation for MIMD Computers*, NISTIR 5571, National Institute of Standards and Technology, Dec. 1994.

[MIN 95] A. Mink, *Operating Principles of the SBus Multikron Interface Board*, NISTIR 5652, National Institute of Standards and Technology, May 1995.

[MIN 97] A. Mink and W Salamon, *Operating Principles of the PCI Bus MultiKron Interface Board*, NISTIR 5993, National Institute of Standards and Technology, Mar. 1997.

[MIN 98] A. Mink, W. Salamon, J. Hollingsworth and R. Arunachalam, *Performance Measurement Using Low Perturbation and High Precision Hardware Assists*, Submitted to the 1998 IEEE Real-Time System Symposium.