WSDLite: A Lightweight Alternative to Windows Sockets Direct Path[‡]

Evan Speight
Computer Systems Laboratory
Cornell University
espeight@csl.cornell.edu

Hazim Shafi
Trilogy Software, Inc.
Austin, Texas
hazim_shafi@trilogy.com

John K. Bennett
Department of Computer Science
University of Colorado at Boulder
jkb@cs.colorado.edu

Abstract

This paper describes WSDLite, a thin software layer that maps a useful subset of the WinSock2 API onto a system area network. The development of WSDLite was motivated by our experience with an early version of Windows Sockets Direct Path (WSDP). WSDP was developed by Microsoft to allow unmodified network applications to exploit the performance and reliability advantages of System Area Networks (SANs). This is accomplished through the use of a software "switch" that, when appropriate, redirects message traffic through the SAN provider protocol stack instead of the standard TCP/IP protocol stack. In addition to the performance advantages, the WSDP architecture offers several other benefits, including automatic support for legacy code, a single well-known API for supporting many different underlying SAN network protocols, and substantially simpler buffer management than that required by the native SAN API. The beta version of WSDP that we examined did not perform as well as expected, achieving only 26% of the native SAN throughput on the system studied. In an effort to determine whether or not this performance difference was intrinsic, we developed WSDLite, a simple alternative to WSDP. WSDLite is a user-level runtime library that implements a small but commonly used subset of the WinSock2 API. For those applications that do not require full WinSock2 functionality, WSDLite provides both the transparency of WSDP and much of the performance benefit of the underlying SAN architecture. In low-level network tests, WSDLite achieves an average of 70% of the native SAN performance. In this paper we describe the design of WSDLite, and present results comparing the performance of both parallel applications and low-level benchmarks using WSDLite, WSDP, TCP, and a native SAN programming library API as the network programming layer.

1. Introduction

System area networks (SANs) are characterized by high bandwidth; low latency (on the order of 10µsec or less for zero-length messages); a switched network environment; reliable transport service implemented directly in hardware; no kernel intervention to send and receive messages; and little or no copying on either the sending or receiving side. SANs may be used for enterprise applications such as databases, web servers, reservation systems, and small to medium scale parallel computing environments.

System area networks have not yet enjoyed wide adoption, in part because of the difficulty associated with writing applications to take advantage of network programming libraries that generally ship with SAN hardware. In order to provide low latency, zero or single-copy messaging between nodes in a SAN, programmers must address a variety of buffer management and flow control issues not typically associated with TCP/IP-style network programming. These issues stem primarily from the use of DMA between the network interface card and the host memory, a process that allows system area networks to provide orders-of-magnitude lower latencies and lower processor utilizations than previous network architectures and protocols. Addressing these requirements can represent a significant burden, not only to programmers developing new applications, but also to those who wish to obtain the benefits of system area networks for the many millions of lines of existing network application code.

To address these concerns, Microsoft, working with SAN implementers, has developed an alternative that will allow network applications to obtain many of the performance benefits associated with system area networks while retaining the familiar programming interface of Berkeley-style sockets in the WinSock2 API. This technology, called *Windows Sockets Direct*

[‡]This research was conducted while all authors were members of the Department of Electrical and Computer Engineering, Rice University, Houston, Texas.

Path (WSDP) [4], fits immediately below the network application and routes network communication calls to either the standard TCP/IP protocol stack or to the WinSock SAN Provider stack, which utilizes the SAN's native network communication mechanism to achieve low latency, high throughput messaging. One of the principal benefits of WSDP is that existing WinSock2-compliant applications do not have to be rewritten, or even recompiled. Currently, WSDP is restricted to use with the Data Center version of the Windows 2000 operating system.

WSDP necessarily implements the entire WinSock2 API, and as a result, incurs overhead costs associated with providing full functionality. In the beta version of WSDP that we have examined, this overhead is quite substantial. While we expect release versions of WSDP software to exhibit better performance than the current beta version, we also believe that there are attractive design alternatives for those applications that do not require full WinSock2 functionality. This paper explores one such alternative.

We have implemented WSDLite, a protocol layer that implements a subset of the WinSock2 API on top of the raw programming interface provided by the GigaNet cLAN implementation of the Virtual Interface Architecture (VIA) [3]. The VI Architecture is the proposed standard for user-level networks developed by Microsoft, Compaq, and Intel. The cLAN architecture provides 9µsec latency for zero-byte messages in our system area network environment when using the VI Programming Library (VIPL) API. WSDLite, similar to WSDP, allows programs written to use TCP/IP to obtain the performance benefits associated with an underlying network architecture that supports VIA. We make use of the Detours [9] binary rewriting software package to intercept the TCP calls implemented by WSDLite and route them to the WSDLite implementation of these functions, while forwarding TCP calls not implemented within WSDLite to the standard WinSock2 protocol stack. Detours allows us to run Winsock2-compilaint applications without recompilation. Unlike WSDP, however, WSDLite only implements a subset (approximately 10%) of WinSock2 functions. The functions implemented were chosen based upon their common use in a variety of software available at our site. A lighter-weight protocol layer such as WSDLite can provide substantial performance benefit relative to full-functioned protocol layers for applications that do not need the full TCP/IP functionality provided by WSDP. Additionally, WSDLite can be used on any Windows NT or Windows 2000 system for which VIA support is available; it is not restricted to Windows 2000 Data Center. We have successfully tested WSDLite on clusters comprised of Windows NT 4.0 workstations and servers, Windows 2000 Professional Workstations, and Windows 2000 Data Center Servers. Simple network latency tests show WSDLite to be an average of 59% faster than the beta WSDP implementation across all message sizes up to 32 Kbytes.

We examine the performance of WSDLite using several network benchmark programs. First, we compare the performance of a series of low-level benchmarks with (1) TCP/IP using WinSock only, (2) TCP/IP using WSDP, (3) TCP/IP using WSDLite, and (4) a version written to use the native VIPL API. For each of the low-level benchmarks, we report roundtrip latency and network throughput. We also report processor utilization, as well as throughput per CPU second, which brings into focus the tradeoff between network and application performance. We next examine the overhead associated with the use of the Detours [9] library to provide Winsock2 transparency. Finally, we use the same four network layer implementations as the messaging layer for the Brazos Parallel Programming Library. By running a set of parallel applications utilizing Brazos, we can evaluate the performance of each network alternative on real applications.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the Virtual Interface Architecture in order to provide the context for the discussion of Windows Sockets Direct Path in Section 3. Section 4 describes the design and implementation of WSDLite. In Section 5 we report the results of our experimental comparison of WSDLite and WSDP. Related work is described in Section 6. We conclude and discuss future work in Section 7.

2. Overview of the VI Architecture

Although Windows Sockets Direct Path is designed to work with a variety of system area network architectures, we are only aware of current WSDP support in the context of the Virtual Interface Architecture. In this section, we present an overview of the VI Architecture as implemented on the GigaNet cLAN GNN1000 network interface card.

Figure 1 depicts the organization of the Virtual Interface Architecture. The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider consists of the VI Network Adapter and a Kernel Agent device driver. The VI Consumer is composed of an application program and an operating system communication facility such as MPI or sockets, although some "VI-aware" applications communicate directly with the VI Provider API. After connection setup by the Kernel Agent, all network actions occur without kernel intervention. This results in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for creation/destruction of VI's, VI connection setup and teardown, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

overhead associated with traditional network protocol stacks, the VI Architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider. This registration process locks down the appropriate pages in memory, which allows for direct DMA operations into user memory by the VI hardware, without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided

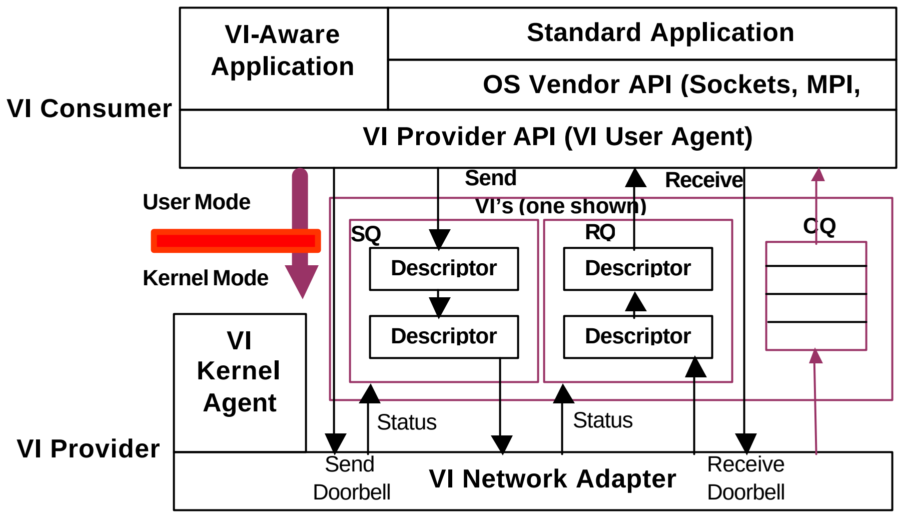

Figure 1. Block Diagram of the Virtual Interface Architecture

A VI consists of a Send Queue and a Receive Queue. VI Consumers post requests (Descriptors) on these queues to send or receive data. Descriptors contain all of the information that the VI Provider needs to process the request, including pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them when completed. VI Consumers remove completed Descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated "Doorbell" that is used to notify the VI network adapter that a new Descriptor has been posted to either the Send or Receive Queue. The Doorbell is directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of Descriptor completions of multiple VI's without requiring an interrupt or kernel call.

2.1. Memory Registration

In order to eliminate the copying between kernel and user buffers that accounts for a large portion of the

to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers, thereby avoiding duplication of locking and translation operations. Memory registration also takes page-locking overhead out of the performance-critical data transfer path.

2.2. Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send and receive semantics, and direct reads and writes to and from the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote node (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. The VI Consumers on both the sending and receiving nodes specify the location of the data. On the sending side, the sending process specifies the memory regions that contain the data to be sent. On the receiving side, the receiving process specifies the memory regions where the data will be placed. The VI Consumer at the receiving end must post a Descriptor to the Receive Queue of a VI before the data is sent. The VI Consumer at the sending end can then post the message to the corresponding VI's Send Queue.

Remote DMA transfers occur using the same descriptors used in send/receive style communication, with the memory handle and virtual address of the remote memory specified in a second data segment of the descriptor. VIA-compliant implementations are required to support remote write, but remote read capability is an optional feature of the VIA Specification. The GigaNet cLAN architecture only provides for remote writes.

3. Windows Sockets Direct Path

Windows Sockets Direct Path (WSDP) allows programs written for TCP/IP to transparently realize the performance advantages of user-level networks such as VIA. Programs developed to the WinSock2 API do not have to be rewritten to take advantage of changes in underlying network architecture to a SAN, nor is recompilation of these programs necessary. This enables legacy network code to work "out of the box" and enjoy at least some benefit of the low message latency associated with SANs. Although WSDP is designed to work with a variety of low-latency SAN architectures, we restrict our discussion here to how WSDP interacts with the cLAN VIA architecture described in Section 2.

WSDP removes many of the pedantic tasks that must be addressed by programs that directly access the VIPL API. These include memory registration, certain aspects of buffer management, and the effort required to port and recompile a sockets-compliant application to use the VIPL API. In the following sections we describe the basic technology associated with WSDP as well as some programming considerations that must be addressed to use WSDP effectively.

Figure 2 depicts a block diagram of the WSDP architecture. The key component of the WSDP architecture is the software switch, which is responsible for routing network operations initiated by WinSock2 API calls to either the standard TCP/IP protocol stack, or to the vendor-supplied SAN WS Provider. In addition to providing access to both of these pathways to the network on an operation-by-operation basis, the switch provides several important functions through the use of a lightweight session executed on top of the SAN provider. This session provides OOB (out of band) support, flow control, and support for the *select* operation. None of these mechanisms are traditionally provided by a typical SAN architecture. There are several operations that require the support of the TCP/IP protocol stack (i.e., do not use WSDP), including:

- Connections to remote subnets.
- Socket creation.
- Raw sockets and UDP sockets - Because SANs support connection-oriented reliable communication, all connectionless and

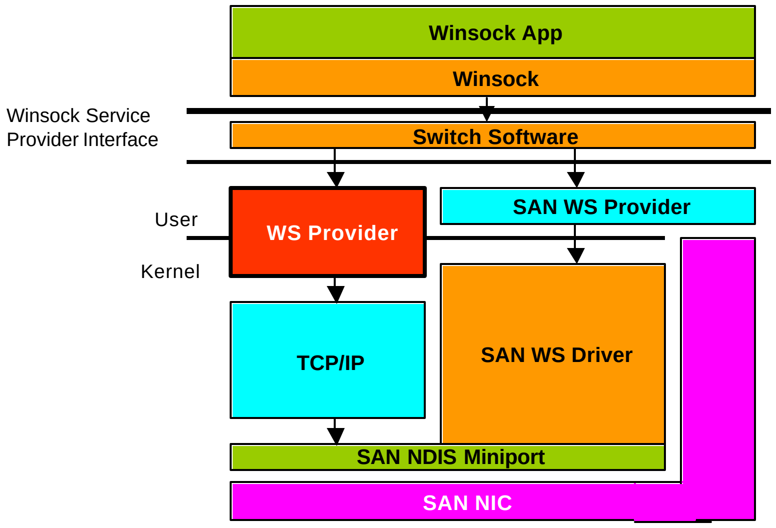

Figure 2. Switch Architecture for Windows Sockets

uncontrolled communication must be handled by the TCP/IP protocol stack. This limits the applicability of WSDP to those applications that (a) use TCP, and (b) do not make use of group communication.

In addition to these restrictions on the use of WSDP, system calls are required to complete most overlapped I/O calls, increasing the latency of these calls due to induced operating system overhead.

The switch component is also responsible for taking care of several programming details that usually must be addressed by the programmer writing directly to the programming library supplied with SANs. A brief discussion of these details follows:

- Buffer registration – As discussed in Section 2.1, buffer space used for messaging must be registered with a SAN provider in order to allow direct DMA into and out of host memory by the NIC. However, there is no provision for this functionality in the WinSock2 specification, as the operating system handles message buffering through copying in a standard WinSock environment. Therefore, the switch component is responsible for ensuring that all buffer regions used for communication are registered with the SAN provider prior to use.

- Buffer placement – Another issue relating to the management of buffers in a system area network requires there to be a buffer posted to a network endpoint prior to receipt of an incoming message. This is again related to the use of DMA between the network interface card and the host memory and the lack of flow control associated with SAN NICs. The switch software pre-posts small buffers to each connection opened through the WS SAN Provider in order to handle incoming messages.

- Support for RDMA – Most system area network include support for remote memory operations, allowing a host node to directly write and/or read data directly from a remote node's address space. No such API exists in the WinSock2 specification. WSDP makes use of the remote write capability of the cLAN architecture in a manner similar to that of WSDLite, as discussed in the next section.

4. WSDLite

WSDLite implements approximately 10% of the WinSock2 API. The following functions are currently implemented by WSDLite: **WSAStartup()**, **WSACleanup()**, **WSASocket()**, **socket()**, **connect()**, **listen()**, **accept()**, **bind()**, **send()**, **WSASend()**, **recv()**, **WSARecv()**, **select()**, **closesocket()**, and **WSAGetLastError()**.

When an application calls a function supported by WSDLite, the function call is intercepted by the Detours [9] runtime library and redirected to the version of the function implemented by WSDLite. In order to leverage functionality existing in the WinSock TCP/IP protocol stack that is not directly related to messaging performance (such as connection procedures and name resolution), some of the WSDLite functions make calls to their WinSock counterparts from within the WSDLite library. For instance, during connection procedures, the WSDLite implementation of **bind()** calls the WinSock2 version of **bind()** internally to check for errors such as two sockets being bound to the same port. In fact, WSDLite duplicates the entire connection process internally on the default TCP/IP protocol stack in order to catch such errors, greatly reducing the code size of the WSDLite implementation.

4.1. Sending Data in WSDLite

When a message is to be sent on a connected pair of sockets, the WSDLite implementation of **WSASend()** or **send()** first must register the buffer containing the data to be sent, if it is not already registered with the cLAN NIC.

Memory Registration Issues

Registering memory is an expensive operation for two reasons. First, registering and deregistering memory on each network access would add unacceptable latency to network operations, especially for small messages. We measured the cost of registering memory for buffer sizes up to 32 Kbytes, and found that it takes roughly 15 µsec to register and deregister a region of memory with the VI Provider, regardless of buffer size. This time increases linearly with buffer size after the size exceeds the 64K segment size used by the NT virtual memory manager. To address this issue, WSDLite maintains a hash table of address ranges that have been used as messaging buffers previously, and this table is consulted before a message can be sent. There are three possible outcomes from the initial hash table lookup:

1. The address has previously been registered, and the size registered is equal or larger than the size of the buffer currently posted. No other action is required.
2. The address has been previously registered, but the size of the region registered does not encompass the entire buffer currently posted. The currently registered region must be deregistered with the NIC and the new region registered.
3. The address has not been previously registered, and WSDLite must register the entire buffer.

To reduce the amount of registering that must be performed by WSDLite, it is important for application programmers to reuse buffers as much as possible.

The second source of overhead associated with memory registration results from the fact that a part of the memory registration process involves pinning messaging buffers into physical memory, which may reduce the resources available for applications. To address this problem, WSDLite employs a simple garbage collection scheme based on timestamps to reclaim unused message buffer space before the amount of pinned RAM impacts application performance.

Choosing the Correct Send Semantic

We have found that minimum latency for messages may be obtained in one of two ways, depending on the size of the message. For small messages, the best performance is achieved by copying data out of temporary receive buffers into the application buffers posted by the corresponding receive operation. For larger messages, lower latency can be achieved by taking advantage of VIA's RDMA capability. When a large message is to be sent, the sending process first sends a setup message to the receiver. This message contains the length of the message to be sent. The receiver registers the memory region to be received into (if it is not already available), and then returns the virtual address and memory region handle to the sending process. The sending process then remote-writes the data directly into the address space of the receiving process, and sends a completion message containing the size of the message written to the receiver when the operation has completed.

The message size at which WSDLite switches from memory copying to RDMA depends on the speed of the host processors, the efficiency of the memory hierarchy, and the latency of network operations.

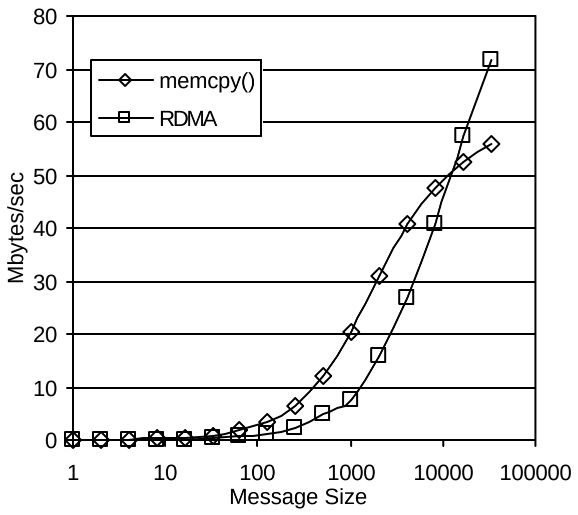

Figure 3. Bandwidth Crossover Point

The crossover point can be clearly seen in Figure 3, which shows the sustainable bandwidth of WSDLite when copying is always used regardless of message size (labeled **memcpy()**), and when RDMA is always used. In the case of our system, the crossover point occurs between 8K and 16K. More precise measurements pinpoint it at 11.9 Kbytes. In general, if copying a memory region of size *n* takes less time than the two additional small messages necessary for the RDMA transfer, memory copying will achieve better performance. Because this value is likely to be different on different machines, WSDLite attempts to automatically determine the optimum value for this cutoff the very first time a socket is created. When the first socket on a machine is created, a small test is run that measures the time to copy regions of memory of varying sizes. When a connection is first made to a remote machine, a test to determine the latency of message sizes corresponding to the setup and acknowledgement messages required for RDMA transfer is also run. The cutoff point for this particular machine can then be determined, and this value is stored in a registry entry that is consulted each time an application makes a connection through WSDLite to a specific remote machine. This step only occurs once during the connection to a remote machine. Subsequent network programs that connect to the remote machine can simply retrieve the cutoff value from the registry based on the remote machine to which the connection is being made. The registry value may be deleted by an administrator at any time to force a recalculation of this parameter, or overridden manually.

4.2. Choice of WSDLite Functions

Finally, we conclude this section with a brief discussion on the functions that we chose to implement in WSDLite. We implemented only those calls that provide the network functionality required by our suite of network programs used for this evaluation. We believe these to be representative of a larger class of network applications that only use basic TCP functionality. By keeping the number of functions small, and the implementation thin, we are able to realize a high percentage of the performance available from the SAN. Many other WinSock2 functions could easily be added to the WSDLite implementation by using our initial functions as a starting point. The downside to our strictly user-level approach is that a different version of WSDLite must be used for each SAN network programming library. However, precisely because we have kept the number of functions both small and basic, this is not a difficult thing to do. The approach taken by WSDP, on the other hand, is one of providing full functionality regardless of the underlying SAN network. This implies that 1) many functions, whose implementations may not easily map to the SAN programming API, will have high overhead; and 2) another level of indirection must exist between the switch software provided by Microsoft and the hardware vendor-provided SAN layer. These two observations necessitate an implementation with higher overhead than a simple user-level library such as WSDLite. Therefore, WSDLite is proposed as a performance alternative to WSDP in certain situations, not a replacement for applications requiring full TCP functionality.

5. Experimental Results

In this section we begin by describing our experimental platform. We then present results comparing several important low-level network performance measurements run under WSDP, WSDLite, TCP, and VIPL on two uniprocessor nodes. Next, we discuss these same measurements when SMP nodes are used. Finally, we conclude the section with results showing the performance of four scientific parallel applications using the four network layer alternatives when run on a larger cluster of SMP servers.

5.1. SAN Configuration

All experiments were performed using a cluster of Compaq Proliant 6400 servers running the Beta 2 release of Windows 2000 Data Center Server, build 2195. Each machine contains one to four 500 Mhz Pentium-III processors, 512 Mbytes of SDRAM, and dual 64-bit PCI busses running at 66 Mhz. The interconnection network is implemented with a single GigaNet GNN1000 NIC in each machine connected via a GNX5000 switch. The switch cut-through latency is 580 ns. The unidirectional latency for a zero-byte message on this system is 9 µsec, and the peak sustainable bandwidth that we have observed is 102 Mbytes/sec.

5.2. Low Level Results

In this section we compare the performance of a message ping-pong test that simply sends messages between two nodes in the cluster. Each node waits for a reply before sending the next message. We compare the performance of this test when using WSDLite, the TCP/IP protocol stack shipped with Windows 2000, WSDP, and the same test written directly to the VIPL API. Note that the first three tests are the same executable; no modifications were necessary when using WSDLite or WSDP to take advantage of the underlying VI hardware. We examine the performance of each of these schemes for message sizes up to 32Kbytes with respect to roundtrip latency, peak sustainable bandwidth, processor utilization, and Mbytes/CPU-second. Finally, we look at the overhead associated with using the Detours [9] package to provide transparent access to WSDLite through the WinSock2 API. Results in this section have been obtained with a single processor in each of the two machines being used. The results of making the same measurements with four processors in each machine is discussed in Section 5.3.

Figures 4 and 5 show the performance of our ping-pong test as measured by roundtrip latency and peak sustainable bandwidth for message sizes from 1 byte to 32 Kbytes. With a single processor in each system, we see that the latency of WSDLite is on average only 19.2% higher than that of native VIPL across all message sizes. The differences between WSDLite and VIPL stem from the extra overhead on each network call of traversing through the TCP-to-VIPL translation layer, the overhead associated with trapping WinSock2 calls using Detours, and the buffer management and flow control that WSDLite must implement.

As expected, TCP performs poorly on latency and peak bandwidth measurements with respect to either WSDLite or VIPL. WSDP performs similarly to TCP, but actually has higher latency at all message sizes and averages 28.8% higher than TCP. The performance of WSDP lags that of WSDLite by an average of 67.9% for all message sizes. This performance advantage of WSDLite is slightly higher at smaller message sizes, with a 69.5% improvement for single-byte messages and a 59.1% improvement for 32Kbyte messages.

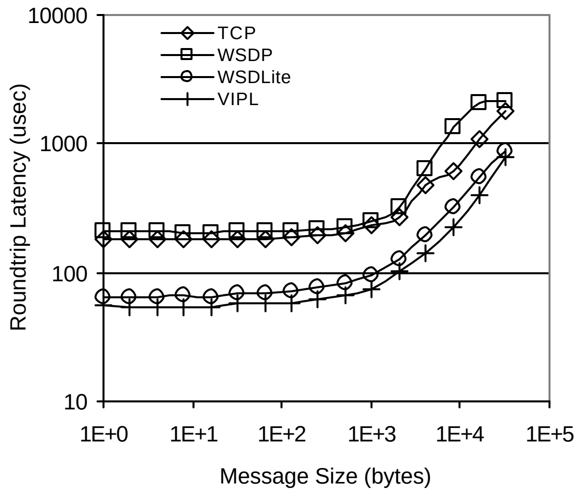

Figure 4. Roundtrip Latency

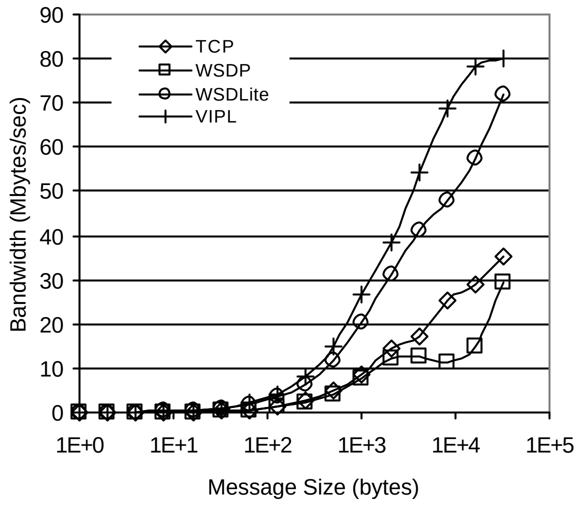

Figure 5. Peak Sustainable Bandwidth

Figure 5 shows that the bandwidth of TCP and WSDP peak at a maximum of around 30-35 Mbytes/sec, whereas VIPL achieves nearly 80 Mbytes/sec, and WSDLite around 72 Mbytes/sec. The performance of WSDLite is restricted below the 16 Kbyte message size from additional copying out of the pre-posted receive buffers, and from the extra setup and acknowledgement messages necessary to implement the RDMA transfer at 16 and 32 Kbyte message sizes. However, these overheads still allow WSDLite to perform within 22% of VIPL. The significantly higher overheads of WSDP caused by multiple software layering and polling between these layers results in performance that is worse than just using TCP directly, regardless of message size.

Figure 6 shows the average processor utilization for the uniprocessor execution of our ping benchmark. For small messages, VIPL has a much higher processor utilization than either of the other three implementations, resulting from a time compression effect due to the small amount of time the message requires "on the wire", and the small fixed costs due to the low overhead of the network protocol. WSDLite and TCP display similar utilizations at small message sizes due to their higher fixed-cost overhead relative to VIPL. WSDP shows the lowest overall utilization for message sizes less than 1K. All implementations that use VI in some layer (WSDP, WSDLite, and VIPL) show low processor utilizations at large message sizes due to the fact that large messages require relatively long DMA times to transfer the message to the NIC hardware, during which time the processor is idle. TCP, on the other hand, buffers and copies messages internally, keeping the utilization high throughout the entire range of message sizes.

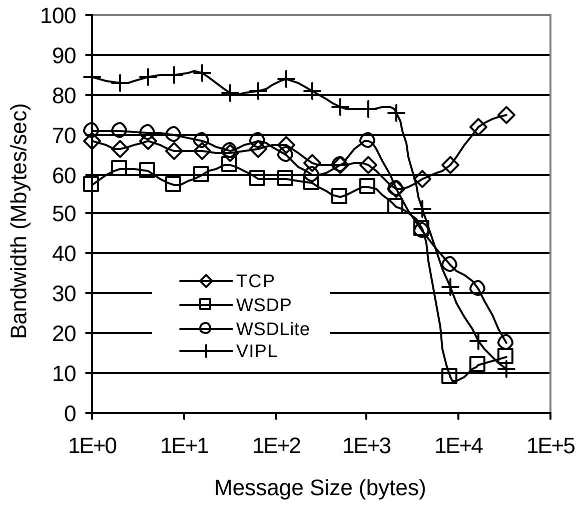

Figure 6. Processor Utilization

The data presented in Figure 6 is misleading, seeming to indicate that WSDP is the most efficient protocol because the processor utilization is lower at smaller message sizes, and the VI Architecture was designed to maximize the performance of small messages [4]. By dividing the peak bandwidth achieved (as presented in Figure 5) by the processor utilization necessary to sustain this bandwidth (as shown in Figure 6), we can track the relative efficiency of a particular network protocol or architecture and find out how much processing time is required to send a fixed amount of data. Figure 7 shows this measurement for the ping test using TCP, WSDP, WSDLite, and VIPL, and is expressed in Mbytes/CPU-second. With only a single processor, TCP and WSDP perform particularly poorly using this metric at small message sizes. The relatively low processor utilization displayed by WSDP in Figure 6 is offset by the extremely low network throughput shown in Figure 5, causing WSDP's performance to nearly mirror that of TCP for message sizes below 8Kbytes.

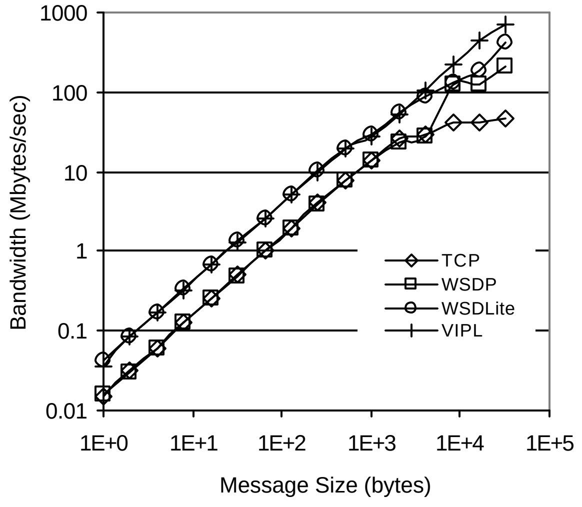

Figure 7. Messaging Efficiency

WSDLite and VIPL, on the other hand, more than make up for the additional processor utilization required with improved bandwidth. Both protocols perform similarly at small message sizes because of the low overhead imposed by the runtime system. This results in a higher amount of data transferred per processor cycle than either WSDP or TCP. As message size increases and the fixed "wire time" becomes a larger portion of the overall network time, the Mbytes/CPU-seconds metric for all protocols increases as the processor overhead becomes less of a factor in overall performance. For very large messages, the performance of the three architectures that utilize VIA begin to converge, whereas the high processor utilization causes the TCP performance to flatten out between 8 and 32 Kbytes.

5.3. SMP Performance

In order to evaluate the performance benefits of running each implementation in an SMP environment, we repopulated each of the two machines used in the experiments with four 500 MHz P-III processors. The performance difference between these results and those presented in Section 5.2 stem from the level of concurrency exploited by the runtime system, as well as the overhead associated with managing threads residing on different processors. Table 1 shows the thread counts present in each process during the execution of the test. Note that the thread counts did not change when moving from a uniprocessor to a 4-way SMP.

	# Threads	Thread Breakdown
TCP/IP	2	1 user thread 1 Winsock thread
WSDP	7	1 user thread 6 WSDP threads
WSDLite	3	1 user thread 1 Winsock thread 1 VIPL thread
VIPL	2	1 user thread 1 VIPL thread

Table 1. Thread Usage

From the thread counts shown in Table 1, we would expect WSDP to exploit concurrency and thus show an improved performance with multiple processors. The other three architectures do not use concurrency in an attempt to reduce overhead. With respect to peak bandwidth, we found that WSDP does indeed perform better with SMP nodes by an average of 17% across all message sizes. The largest improvement occurred at 16K messages (48%). Because neither WSDLite nor VIPL use concurrency within the runtime system, the performance of these two implementations remains nearly constant regardless of the number of processors available (average improvement of 4.5% and .6%, respectively). However, the throughput of WSDLite remains an average of 67% better than that of WSDP across all message sizes.

5.4. Overhead Associated with Detours

Finally, we examine the performance impact of using Detours to eliminate the necessity of recompiling a WinSock2 application to use the WSDLite library. Detours instruments x86 binaries and inserts jump calls to trap targeted Win32 functions. We have configured Detours to trap all of the calls implemented in the WSDLite library and redirect them to the WSDLite implementation of the functions. Using this redirection, as opposed to recompiling the program and linking directly with the WSDLite library, incurs a 3 μsec overhead per roundtrip message. It may be possible to reduce this further through more direct interception methods.

5.5. Application Results

In this section we examine the performance of WSDP, WSDLite, TCP, and VIPL when each is used as the underlying network layer for the Brazos Parallel Programming Environment [12]. Brazos provides transparent shared memory and message passing support across a network of SMP machines running Windows 2000/NT. Brazos was originally developed for use with UDP on WinSock, delivering superior performance to distributed shared memory applications. Support for VI was later added. For the purposes of this study, we converted Brazos to run using TCP sockets and present results for four shared memory scientific applications running on two quad Compaq Proliant 6400 servers. The four applications include Ilink, a genetic linkage program used to trace genes through family histories; Barnes Hut, an n-body problem solver from the SPLASH-2 benchmark suite [15]; LU decomposition, also from SPLASH-2; and FFT-3D, used to solve fast Fourier transforms in three dimensions, from the NAS parallel benchmark suite [1].

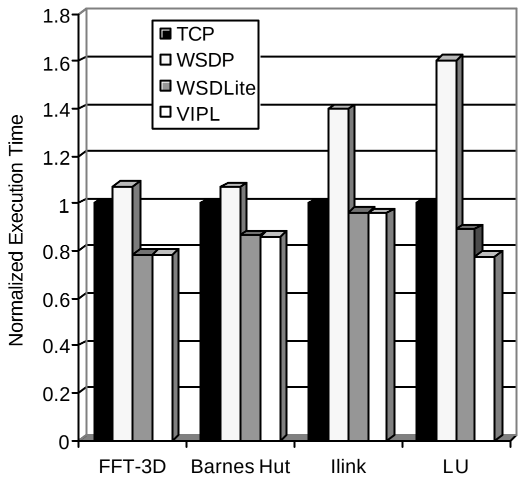

Figure 8. Parallel Application Performance

Figure 8 shows the performance of these four applications on WSDLite, WSDP, and VIPL in terms of the execution time normalized to that of the execution time when run on TCP. For three of the four applications (FFT-3D, Barnes Hut, and Ilink), WSDLite performs within 2% of the VIPL performance, demonstrating the low overhead associated with the WSDLite runtime protocol layer. For LU, the performance of WSDLite suffers slightly due to send throttling in the WSDLite protocol, causing some send operations to stall waiting for available buffers to be re-posted on the receiving node.

For these experiments, WSDP performs particularly poorly relative to TCP. We believe this performance degradation to be the result of processor contention due to the high number of threads used in the WSDP protocol stack (see Table 1). Threading in WSDP is used to boost concurrency between the software layers that make up the protocol stack. Synchronization and polling between these layers apparently results in processor starvation for computation threads, leading to a potentially large increase in parallel execution time.

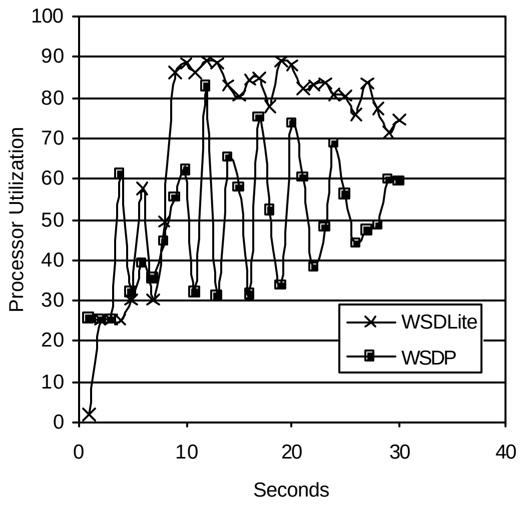

Figure 9. Processor Utilization for LU

Ilink and LU exhibit this effect to a larger degree because the computation-to-communication ratio of these applications is higher than that of FFT-3D or Barnes Hut. Therefore the contention for available processing resources is higher in these two applications, and the extra threads in WSDP exacerbate the problem. Figure 9 shows the average processor utilization of the four processors on one node when LU is run using WSDLite and WSDP. Data is shown for the first 30 seconds of the program's execution time, which represents the entire program execution under WSDLite. As indicated, the processor utilization for WSDLite remains high throughout the program's execution, resulting in a high parallel speedup for this application. When using WSDP, the processor utilization varies widely during the course of execution as the computation threads compete with the threads that implement WSDP. The resulting context switching reduces the effective processor utilization to the varying levels shown in Figure 9, and results in a 60% increase in overall execution time.

6. Related Work

Previous work in this area can be divided roughly into two categories: new protocol and network interface designs, and attempts to deliver the improvements of these new network protocols and interfaces to applications. Our work falls into the second category, thus we concentrate on related work in this area.

Windows Sockets Direct Path (WSDP) [4] attempts to deliver the performance of user-level network interfaces (VIA in our case) transparently to TCP/IP networked applications written for the WinSock2 API. The approach taken by WSDLite differs from WSDP in two significant ways. First, WSDLite only implements a subset of the full WinSock2 API, albeit a useful subset that suffices for many networked applications. Second, our technique is not directly transparent. Although we do not require access to source code nor recompilation, we have to modify the applicable binaries using Detours in order for WinSock2 calls to be redirected. This is achieved by simply running the desired executable with a program called *withdll*, which injects the WSDLite DLL into the executable and rewrites the binary file to cause the redirect to the WSDLite implementation of WinSock2 functions.

VIA derives from a large body of related work in user-level communication, with the basic operation coming out of the U-Net research by von Eicken et al. [7]. As part of the U-Net research, a proof-of-concept implementation of TCP/IP was developed that delivered close to the raw performance of U-Net to TCP- and UDP-based applications. The results of this implementation, presented by the authors in [7], were partially what led us to investigate a performance alternative to the beta version of WSDP that we initially examined. VIA draws from several other research projects including application device channels [5], which provide the model for virtual interfaces to the network; and Virtual Memory Mapped Communication (VMMC) [6] and Active Messages (AM) [8], which provide the model for remote memory operations used in VIA. Other projects with similar goals to WSDLite and WSDP include Fast Sockets [11], which like WSDLite offers increased communication performance by collapsing protocol layers, using simple buffer management strategies, and by using “receive posting” to bypass data copying. Thekkath et al. proposed separating network control and data flow, and employed unused processor opcodes to implement remote memory operations [13]. Fast Messages [10] allow direct user-level access to the network interface, but do not support simultaneous use by multiple applications. The HP Hamlyn network implements user-level sends and receives in hardware [2]. ParaStation [14] provides unprotected user-level access to the network interface. With Active Messages [8], each message contains the address of a user-level handler that is executed upon message arrival with the message body as an argument. This allows the programmer and compiler to overlap communication and computation, thereby hiding latency.

7. Conclusions and Future Work

For those applications that use only the WSDLite subset of TCP functionality, we have demonstrated that WSDLite offers significant performance advantages relative to WSDP. However, this result must be qualified in several ways. First, we are using a beta implementation of WSDP. We expect the performance of subsequent versions of WSDP to improve. Second, some users may consider the modification of application binaries required by WSDLite in order to achieve transparency to be too aggressive for comfort. Third, while it is relatively easy to add additional functionality to WSDLite, certain aspects of Winsock2 functionality would likely be difficult to implement without incurring additionally overhead. In spite of these acknowledged limitations, WSDLite provides a useful tool for many applications.

We will continue to update our results as new versions of WSDP and the cLAN Winsock provider become available. We also intend to experiment with additional network applications. We are currently evaluating FTP and a web-based client/server database application for this purpose.

8. Acknowledgements

The authors would like to thank Galen Hunt of Microsoft Corporation for providing valuable information and sample code that allowed us to successfully use the Detours package with WSDLite.

This research was supported in part by grants and assistance from Compaq Computer Corporation, GigaNet Corporation, Intel Corporation, Microsoft Corporation, and by the Texas Advanced Technology Program under Grant No. 003604-022.

Bibliography

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, The NAS Parallel Benchmarks, NASA Ames RNR-91-002, August 1991.

[2] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pp. 245-259, 1996.

[3] Compaq Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture Specification, Version 1.0. 1997.

[4] Microsoft Corporation. *Windows Sockets Direct Path for System Area Networks*. Microsoft Corporation, 2000.

[5] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th Annual Symposium on Operating System Principles*, pp. 189-202, 1993.

[6] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the International Parallel Processing Symposium*, pp. 388-396, 1997.

[7] T. V. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 40-53, December 1995.

[8] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 256-266, 1992.

[9] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[10] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, 1995.

[11] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the Usenix 1997 Conference*, January 1997.

[12] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, pp. 95-106, August 1997.

[13] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, October 1994.

[14] T. M. Warschko, J. M. Blum, and W. F. Tichy. The ParaPC/ParaStation Project: Efficient Parallel Computing by Clustering Workstations. University of Karlsruhe, Department of Informatics Technical Report 13/96, 1996.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.