Sockets Direct Protocol for Hybrid Network Stacks: A Case Study with iWARP over 10G Ethernet

P. Balaji, S. Bhagvat, R. Thakur, and D. K. Panda

Preprint
Argonne National Laboratory #

Sockets Direct Protocol for Hybrid Network Stacks: A Case Study with iWARP over 10G Ethernet*

P. Balaji[1], S. Bhagvat[2], R. Thakur[1], and D. K. Panda[3]

[1] Math. and Comp. Sci., Argonne Natl. Lab, {balaji,thakur}@mcs.anl.gov
[2] Scalable Systems Group, Dell Inc., sitha_bhagvat@dell.com
[3] Computer Science and Engg., Ohio State University, panda@cse.ohio-state.edu

Abstract. As high-end computing systems continue to grow, the need for advanced networking capabilities, such as hot-spot avoidance and fault tolerance, is becoming important. While the traditional approach of utilizing intelligent network hardware has worked well to achieve high performance, adding more and more features makes the hardware complex and expensive. Consequently, protocol stacks such as iWARP and MX for 10-Gigabit Ethernet and QLogic InfiniBand, utilize hybrid hardware-software designs that take advantage of the processing power of multi-core processors together with network hardware accelerators. However, upper-layer stacks on these networks, such as the Sockets Direct Protocol (SDP), have not kept pace with such shift in paradigm, and have continued to assume complete hardware offload, leading to redundant features and performance loss. In this paper, we propose an enhanced design for SDP that allows network stacks to specify components implemented in hardware and software, and uses this information to optimize its execution.

1 Introduction

As high-end computing (HEC) systems continue to increase rapidly in size, their communication subsystems must scale as well. For large-scale systems, in addition to raw performance, advanced communication features such as capability to avoid hot-spot congestion [29, 33] and hardware faults [15] are also becoming increasingly important. While the traditional approach of utilizing intelligent hardware support on the network adapters (e.g., Mellanox InfiniBand [2], Myrinet 2000 [14], Quadrics [28], hardware iWARP [19, 23]) has worked well to achieve high performance, adding more and more features makes the hardware complex, error prone, and expensive.

At the same time, there have been prominent advances in processor technology, especially powered by the advent of multi-core architectures [25, 5]. Thus, to take advantage of these two trends, several network stacks (e.g., QLogic InfiniBand [30], Myri-10G [27], software iWARP [8]) have started to utilize hybrid

* This work was supported in part by the National Science Foundation Grant #0702182 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

hardware-software stack designs (known as *hybrid network stacks*). These hybrid network stacks take advantage of the processing power of multi-core processors together with network hardware accelerators to achieve high performance while providing the flexibility to add most communication features relevant to modern HEC systems.

However, several upper-layer stacks on top of these networks have not been able to keep pace with such shift in paradigm of network communication stacks. For example, existing implementations of high-performance sockets on high-speed networks, such as the Sockets Direct Protocol (SDP) [10] over InfiniBand (IB) [24] and 10-Gigabit Ethernet (10GE) iWARP [31], continue to assume complete hardware offload. Consequently, they perform various tasks, such as data buffering to optimize small message communication and message-level flow-control that allow them to achieve high performance on hardware-offloaded network stacks but are redundant on hybrid network stacks and can add significant performance overheads.

In this paper, we perform a case study with SDP on top of a hybrid hardware-software iWARP design over 10GE, and study the drawbacks of its existing implementation. We also propose an enhanced design for SDP that allows network stacks to specify what components are implemented in hardware and what are implemented in software, and uses this information to avoid redundancy in the overall stack. We experimentally compare our proposed approach with the traditional design of SDP using both micro-benchmarks as well as two real applications (virtual microscope [17] and iso-surface oil-reservoir data visualization [13]) built on top of the DataCutter library [12]. Our results demonstrate that the proposed approach can outperform the traditional approach by nearly 20% in micro-benchmarks and about 5% in real applications.

2 Background

In this section, we present a brief overview of SDP and iWARP implementations.

2.1 Overview of SDP

SDP is a byte-stream transport protocol that closely mimics TCP sockets' stream semantics. It is an industry-standard specification for IB and iWARP that utilizes advanced capabilities provided by the network stacks to achieve high performance without requiring modifications to existing sockets-based applications. SDP is layered on top of IB or iWARP's message-oriented transfer model. The mapping of the byte-stream protocol to the underlying message-oriented semantics was designed to transfer application data by one of two methods: through intermediate private buffers (using buffer copy) or directly between application buffers (zero copy).

Zero-copy Approach: Hardware-offloaded protocol stacks allow zero-copy communication of application data. However, such communication comes with several restrictions. For instance, communication buffers have to be *registered*: (i) they need to be pinned so that their physical memory pages cannot be swapped out and (ii) the virtual-to-physical address translation must be provided to the communication stack to potentially be cached on the network adapter. Also, to perform zero-copy communication in SDP, the sender and the receiver have to synchronize on the source and destination buffers, which adds overhead. Thus, while zero-copy communication avoids memory copies, it adds other overheads. Accordingly, SDP uses it only for transferring large messages.

Buffer-copy Approach: Due to the overheads of zero-copy communication, SDP utilizes a buffer-copy approach for small messages. In this approach, it pre-registers private buffers at connection-establishment time. On a send, the data is copied into the registered private buffers, communication carried out from and to these buffers, and finally the data copied out to the destination application buffer on the receiver side. However, the buffer-copy approach also comes with two disadvantages. First, data that needs to be communicated has to be copied on the sender and receiver side. Second, since the number of the private registered buffers is limited, the sender has to perform flow-control to make sure the receiver buffers are not overrun. SDP uses the buffer-copy approach only for transferring small messages to avoid being penalized by the message-copy overheads.

2.2 Overview of iWARP

The Internet Wide Area RDMA Protocol (iWARP) is a new initiative by the Internet Engineering Task Force (IETF) [1] and the Remote Direct Memory Access Consortium (RDMAC) [3]. The iWARP standard, when offloaded on to the network adapter, provides two primary extensions to regular Ethernet: (i) it exposes a rich interface including zero-copy, asynchronous and one-sided communication primitives and (ii) it internally relies on an implementation of the TCP/IP stack to allow such communication while maintaining backward compatibility with existing TCP/IP. iWARP comprises three protocol layers atop TCP/IP: (i) RDMAP verbs, (ii) Remote Direct Data Placement (RDDP) protocol and (iii) Marker PDU Aligned (MPA) protocol.

RDMAP verbs [6] is a thin interface that allows applications to interact with RDDP. RDDP provides reliable, in-order delivery using a reliable IP based protocol such as TCP. It distinguishes iWARP from other high-speed network stacks based on its capability to decouple data placement and message delivery; that is, even if packets arrive out-of-order, RDDP directly places them in the appropriate location of the final destination buffer (data placement), and the upper layer is informed about the placement of the data only after the entire message is placed (data delivery). This, of course, assumes that RDDP can correctly identify and understand the contents of out-of-order TCP/IP packets. The Marker PDU Aligned (MPA) protocol provides RDDP with the necessary support for achieving this.

Switches that support splicing [18] (e.g., firewalls and port-forwarding switches) can cause *middle box fragmentation*, i.e., packets going into the switch can be segmented into multiple packets or multiple packets can be coalesced into a single packet. This makes it impossible for the end node to recognize the RDDP headers without additional information if packets arrive out of order. To tackle

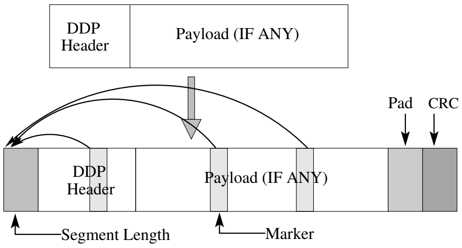

Fig. 1. MPA Protocol Frame

this problem, iWARP uses MPA [20]. The MPA frame format, referred to as a Framing Protocol Data Unit (FPDU), is represented in Figure 1. Apart from additional headers and footers, the FPDU introduces strips of data, known as *markers*, that are spaced uniformly based on the TCP sequence number. These *markers* always point to the RDDP header and provide the receiver with a deterministic way to find them. When a packet arrives out-of-order, it can use these markers to identify the *start* of the iWARP frame and, using that, the rest of the fields.

3 Hybrid Hardware-Software iWARP Stack

Several different implementations of iWARP exist, including complete software implementations [9, 21], complete hardware implementations [19] and hybrid hardware-software implementations [8]. In general, hardware implementations are optimized for performance but do not offer many advanced features; software implementations tend to be more feature complete with respect to their capability to efficiently handle out-of-order communication, packet drops, etc., but do not provide the best performance. The hybrid hardware-software implementation takes the best of both worlds by achieving high performance using network hardware accelerators, while still providing the advanced features using the capabilities of host processors. In this section, we present a high-level description of our previous work on a hybrid hardware-software iWARP stack [8].

The iWARP protocol layers perform various tasks corresponding to data ordering, data integrity, connection management, and backward compatibility. Of these, three tasks are of particular importance as they can heavily impact the performance of the stack: (i) CRC-based data integrity, (ii) connection demultiplexing, and (iii) placement of markers.

CRC is easily the most compute intensive task in the iWARP stack. There have been several attempts to improve its performance [32, 16], often at the cost of additional memory usage. However, its computational overhead is still considered to be very high [26]. Thus, a complete software implementation can be heavily impacted by this overhead.

Traditional TCP/IP performs demultiplexing (DEMUX) of packets in host-space, i.e., the NIC hands over all packets to the host and the host identifies the connection to which each packet belongs and places it in the appropriate queue. While this is not a major concern for applications that only deal with a single (active) connection, this introduces significant overheads for applications dealing with several connections simultaneously (e.g., cache thrashing and CPU interruption for non-critical data). Again, doing this in software is not the ideal solution either.

Placement of markers is one of the trickiest components in the iWARP stack. Since the markers have to be inserted within the data stream, data has to be moved to achieve this. In a software implementation of iWARP, this is done by performing an additional copy of the data. This task is difficult to implement efficiently in hardware without using true scatter/gather DMA engines, which are not commonly available (most DMA engines provide a scatter/gather DMA API, but internally perform individual DMA operations). Thus, hardware iWARP achieves sub-optimal performance for this component [8].

Hybrid iWARP, behaves like software iWARP for the placement of markers (that is, it does this by performing an additional data copy), while using hardware accelerators for the remaining tasks (such as CRC and DEMUX). Thus, in summary, software iWARP performs everything in software, hardware iWARP performs everything in hardware, and hybrid iWARP performs everything in hardware except the placement of markers, which is done in software using an extra buffer copy.

4 SDP for Hybrid Hardware-Software Network Stacks

As briefly described in Section 2.1, existing designs of SDP have been heavily optimized for hardware offloaded protocol stacks. However, such designs are often not the best when utilized on hybrid network stacks. In this Section, we present a few sample existing designs that perform sub-optimally on hybrid iWARP network stacks, and propose enhancements that can improve their performance.

4.1 Redundant Buffer Copy

SDP performs data buffering for small messages. Such buffering has several advantages on hardware-offloaded network stacks including avoiding registration cost, and avoiding synchronization between the sender and receiver. However, on hybrid network stacks, these designs are redundant. For example, the hybrid iWARP stack internally performs data buffering before communication while handling the placement of markers in software. Thus, buffering at both layers is not required and causes performance overhead.

However, avoiding such redundancy is not trivial. Buffering performed within the iWARP stack allows the iWARP implementation to add markers within the data stream; data is copied such that small gaps are left open where the markers can be placed once the copy is complete. On the other hand, buffering within the SDP implementation allows it to handle the socket stream semantics where one large message sent by the sender can be read as multiple small messages by the receiver. Since iWARP follows message-based semantics, it does not allow for such capabilities. Thus, both stacks have specific purposes for buffering that cannot be ignored.

In our approach, we allow the SDP and iWARP stacks to have integrated data buffering. Specifically, the SDP stack performs buffering, but does so in a manner that is compatible with iWARP's buffering. That is, it copies data while leaving small gaps based on the TCP sequence numbers of the data (retrieved from the iWARP stack). The iWARP stack uses this buffering performed by SDP and adds the markers in-place directly in the SDP buffers. While this approach requires close interaction between the SDP and iWARP stacks, and thus loses some amount of generality of the SDP stack, it can reduce the amount of buffering required and thus improve performance.

4.2 Protocol Interface Extensions for Message Coalescing

Message coalescing has been shown to achieve high performance by reducing the number of I/O bus and network transactions required for transferring data [7]. However, it is quite difficult to achieve in hardware-offloaded protocol stacks owing to the hardware-design complexity and resource requirement associated with such a design. For hybrid network stacks, on the other hand, this might not be a concern when implemented in software using the host-memory resources. The issue, however, is that most protocols (including iWARP) do not provide any interface that allow upper layers (such as SDP) to coalesce multiple messages before sending them out on the wire. Further, message coalescing inherently suffers from issues of performance loss in cases where the sender process buffers data hoping to coalesce it with more later arriving data, while the receiver process waits for the message to be transmitted by the sender.

To solve this problem, we extended the interface provided by the hybrid iWARP implementation to allow upper layers to "append" a new message to a previously queued message whose communication has not yet been initiated. Specifically, since hybrid iWARP implementations perform flow control, communication requests that have been handed off to them might not be initiated immediately. Therefore, a later initiated communication request can append itself to this message. This approach has multiple advantages. First, multiple small messages that are being communicated in a short interval can be coalesced into one large message, thus reducing the number of network transactions and improving performance. Second, this approach does not cause any loss of performance as compared to a non-coalescing approach, since data is coalesced only when the previous message was already waiting to be sent out due to flow control; that is, a message is never artificially held back hoping to be coalesced with a later arriving message. Third, this approach reduces the number of iWARP headers that are sent out on the network since coalesced messages are sent out with one header as one single message. This is a big gain for small messages, where the iWARP header forms a major fraction of the total frame size.

4.3 Asynchronous Flow Control

Traditional implementations of SDP over hardware-offloaded iWARP perform explicit flow control. That is, if there are no credits to send data out, the sender copies the data into the temporary private buffers and *waits* for more credits to arrive (similar to advertised window in TCP). However, for hybrid iWARP implementations, such flow control is redundant since the iWARP implementation itself performs flow control as well. Furthermore, the iWARP flow control is more sophisticated as it is implemented within the kernel and uses light-weight hardware interrupts to perform asynchronous progress. Thus, in our approach, we completely disable SDP-level flow control and only rely on iWARP flow control.

While this approach works well for synchronous sockets, for asynchronous sockets, it has the drawback of its inability to call application-specific call-back functions. That is, asynchronous sockets (such as those used in Windows) allow applications to specify call-back functions that are triggered when a message send or receive is completed. To allow for such functionality, we extended the iWARP interface to specify such details, including call-back functions and message send/receive watermarks (that is, at what point the call-back should be triggered). Again, while such functionality would be extremely cumbersome and difficult to implement on hardware offloaded network stacks, it is relatively straightforward on hybrid network stacks.

5 Experimental Results and Analysis

In this section, we first evaluate our proposed approach with the latency and bandwidth micro-benchmarks in Section 5.2. We study the cache misses caused by existing approaches and how our approach reduces them in Section 5.3. Finally, we evaluate two real applications comparing our proposed approach with existing approaches in Section 5.4.

5.1 Experimental Testbed

For our experiments, we used a 4-node cluster built around SuperMicro SUPER X5DL8-GG motherboards with ServerWorks GC LE chipsets, which include 133-MHz PCI-X interfaces. Each node has two Intel Xeon 3.0 GHz processors with a 512-KB cache, a 533 MHz front-side bus and 2 GB of 266-MHz DDR SDRAM. The nodes are connected with Chelsio T110 10GE TCP offload engines through a 12-port Fujitsu XG800 switch. The software stack on the machines is based on linux-2.4.22smp and RedHat linux distribution. The driver version on the NICs is 1.2.0. For each experiment, ten or more runs/executions are conducted, the highest and lowest values dropped (to discard anomalies) and the average of the remaining values is reported. For micro-benchmark evaluations, the results of each run are an average of 10,000 or more iterations.

5.2 Micro-Benchmark Evaluation

Ping-pong Latency: Figure 2(a) compares the ping-pong latency of traditional SDP with our new approach. In this experiment, the sender sends a message of size S to the receiver. On receiving this message, the receiver sends back another message of the same size to the sender. This is repeated several times and the

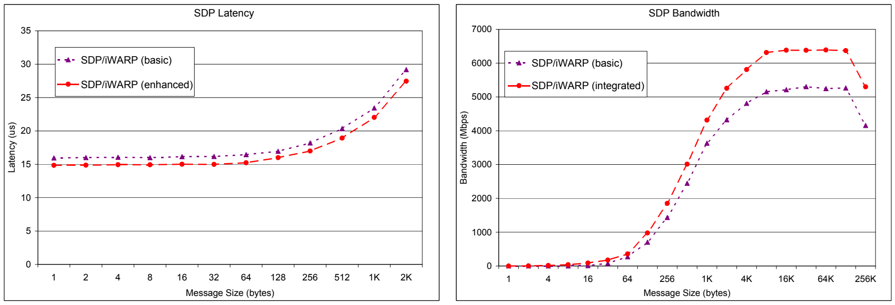

Fig. 2. SDP Performance: (a) Latency and (b) Bandwidth

total time averaged over the number of iterations, which gives the average round-trip time. The ping-pong latency reported here is one half of the round trip time, i.e., the time taken for a message to be transferred from one node to another.

As shown in the figure, our proposed approach (SDP (enhanced)) outperforms traditional SDP (SDP (basic)) by about 10%. This is attributed to several reasons including the reduced buffer copies, and lack of redundant flow-control.
Unidirectional Bandwidth: Figure 2(b) shows a comparison of the unidirectional bandwidth. In this experiment, the sender sends a single message of size S a number of times to the receiver. On receiving all the messages, the receiver sends back one small message to the sender informing that it has received the messages. The sender calculates the total time, subtracts the one-way latency of the message sent by the receiver, and based on the remaining time, calculates the amount of data it had transmitted per unit time.

As shown in the figure, our proposed approach outperforms traditional SDP by about 20% in this case. This behavior is expected as, for large messages, traditional SDP gets significantly hurt by the additional buffer copy and loses performance. Furthermore, as we will see in Section 5.3, its performance is further affected by secondary issues such as increased cache misses.

5.3 Cache-Miss Analysis

Figure 3 shows the analysis of cache-to-network traffic ratio, comparing traditional SDP to our proposed approach; that is, how many bytes of data have to be fetched to or flushed from cache, for every byte of data sent on the network. We see that traditional SDP requires nearly four bytes of cache traffic for every byte of network traffic, as compared to our approach that requires only two.

Specifically, in the bandwidth micro-benchmark that we used, all messages are sent from the same application buffer, but the SDP and iWARP private buffers are used from a circular queue. Thus, the application buffer is always in cache, but the private buffers are never in cache. When the application data is copied to the SDP buffer, the SDP buffer needs to be fetched into cache. Next, when the data is copied from the SDP buffer to the iWARP buffer, the iWARP buffers needs to be fetched into cache. Finally, when the next set of buffers are fetched, both the SDP and iWARP buffers have to be flushed out of cache, since

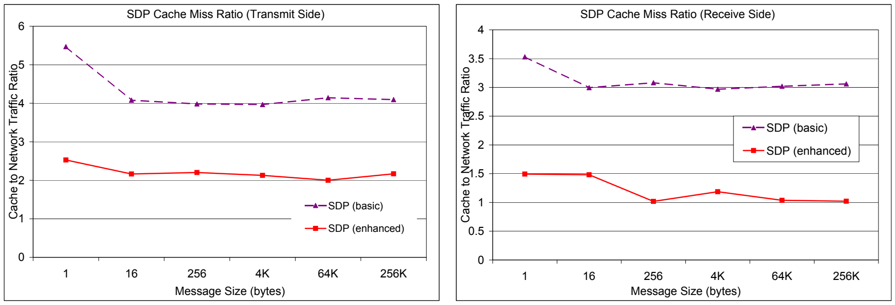

Fig. 3. SDP Cache to Network Traffic Ratio: (a) Transmit and (b) Receive

they are both dirty. Thus, there are two bytes fetched to cache and two bytes flushed from cache (total of four bytes of cache traffic), for every byte of data sent over the network. For our proposed approach, on the other hand, since the SDP/iWARP buffer is combined, only this combined buffer needs to be fetched into cache and flushed out from there, for a total of two bytes of cache traffic per network byte.

On the receive side (Figure 3(b)), the analysis is similar. For traditional SDP, when the data arrives, it is directly DMA'ed into the iWARP private buffer. When the data is copied to the SDP private buffer, both the iWARP and SDP private buffers need to be fetched to cache. Since the same application buffer is used throughout the experiment, it can be expected to stay in cache. However, since the SDP buffer is dirty it has to be flushed out of cache when the next set of buffers are fetched in. Thus, there are two bytes of data fetched and one byte of data flushed for every byte of data sent over the network. For our proposed approach, the combined SDP/iWARP buffer has to be fetched to cache to copy into the application buffer, i.e., one byte of cache traffic per network byte. Note that this buffer does not need to be flushed since it was never dirtied after fetching to cache.

5.4 Application-Level Evaluation

In this section, we evaluate our proposed approach based on two different applications, virtual microscope [17] and iso-surface visual rendering [13], that have been developed using the DataCutter library [11].

Overview of the DataCutter Library: DataCutter is a component-based framework [12] developed at the University of Maryland. It provides a framework, called filter-stream programming, for developing data-intensive applications. In this framework, the application-processing structure is implemented as a set of components, called *filters*. Data exchange between filters is performed through a *stream* abstraction that denotes a unidirectional data flow from one filter to another. The overall processing structure of an application is realized by a *filter group*, which is a set of filters connected through logical streams. An application query is handled as a *unit of work* (UOW) by the filter group. The size of the UOW also represents the granularity in which data segments are distributed in the system and the granularity in which data processing is pipelined. Several data-intensive applications have been designed and developed by using the DataCutter run-time framework, such as the virtual-microscope application and the iso-surface visual-rendering application.

Virtual Microscope: Virtual microscope [17] is a digitized microscopy application. The software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope is a challenging issue [4, 17]. The main difficulty stems from the handling of large volumes of image data, which can range from a few hundreds of megabytes to several gigabytes. At a basic level, the software system should emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing of client queries requires projecting high-resolution data onto a grid of suitable resolution and appropriately composing pixels mapping onto a single grid point.

Iso-surface Visual Rendering: Iso-surface rendering [22] is a widely used technique in many areas, including environmental simulations, biomedical images, and oil reservoir simulators, for extracting and simplifying visualization of large datasets within a 3D volume. In this paper, we utilize a component-based implementation of such rendering [13].

Evaluating the Applications: Figure 4 shows the performance of the virtual microscope and iso-surface visual-rendering applications for the different SDP designs. The applications were executed with a UOW of 1KB and 8KB, respectively. The complete dataset is about 1 GB in size and is hosted on a *RAM disk* in order to avoid disk fetch overheads in the experiment. The virtual-microscope application used five filters: *read data*, *decompress*, *clip*, *zoom*, and *view*. The iso-surface visual-rendering application used four filters: *read dataset*, *iso-surface extraction*, *shade and rasterize*, and *merge/view*. Each filter performs some computation and communicates the processed data to the next filter. Once the communication is initiated, the filter starts computation on the next UOW, thus attempting to overlap communication with computation.

For the virtual-microscope application, as shown in Figure 4(a), our proposed approach outperforms traditional SDP by nearly 5%. This benefit is mainly attributed to the benefits of message coalescing. Since the UOW size used in this application is quite small, the buffer-copy overhead would not be too high. Similarly, since after coalescing, the number of messages is fewer, running out of buffer credits happens rarely, and hence flow-control does not play a major role either.

As shown in the Figure 4(b), for the iso-surface application, our proposed approach outperforms traditional SDP by more than 5%. This benefit is attributed to mainly the reduction in buffer copies and the lack of redundant flow-control. Message coalescing would likely have little effect since the virtual microscope application uses about 8KB data chunks (UOW is 8KB), where the bandwidth is already close to the peak and coalescing would not help it much. Also, DataCutter relies only on synchronous sockets, so asynchronous sockets optimizations would not help either.

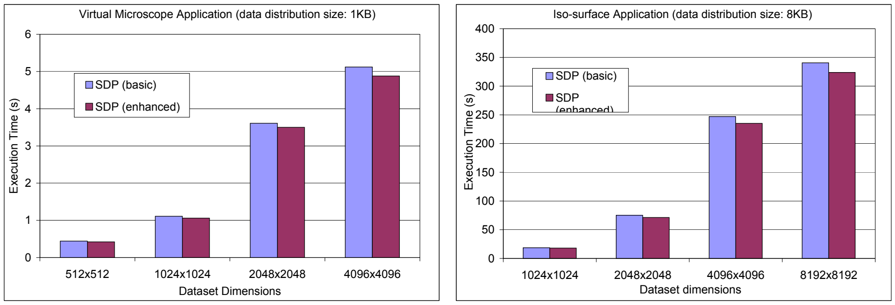

Fig. 4. Application Performance: (a) Virtual Microscope and (b) Iso-surface Oil Reservoir Data Visualization

6 Conclusions and Future Work

In this paper, we proposed an extended design for SDP that uses information on which components of the network protocol stack are implemented in hardware and which are implemented in software to optimize its execution. We compared our proposed approach with existing implementations and showed that we can achieve significant performance improvements. As a part of our future work, we would like to study such enhancements in other protocol stacks, including MPI, as well. Furthermore, we would like to generalize our model so that all upper-layer protocols can query for which components are implemented in hardware and software in a uniform manner.

References

1. IETF. http://www.ietf.org.
2. Mellanox Technologies. `http://www.mellanox.com`.
3. RDMA Consortium. http://www.rdmaconsortium.org.
4. A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital Dynamic Telepathology - The Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.
5. AMD Quad-core Opteron processor. http://multicore.amd.com/us-en/quadcore/.
6. S. Bailey and T. Talpey. Remote Direct Data Placement (RDDP), April 2005.
7. P. Balaji, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp. Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand. In *ICPP*, 2007.
8. P. Balaji, W. Feng, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp. Analyzing the Impact of Supporting Out-of-Order Communication on In-order Performance with iWARP. In *SC*, 2007.
9. P. Balaji, H. W. Jin, K. Vaidyanathan, and D. K. Panda. Supporting iWARP Compatibility and Features for Regular Network Adapters. In *RAIT*, 2005.
10. P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *ISPASS '04*.

11. M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *HCW*, 2000.
12. M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed Processing of Very Large Datasets with DataCutter. *Parallel Computing*, October 2001.
13. M. D. Beynon, T. Kurc, U. Catalyurek, and J. Saltz. A Component-based Implementation of Iso-surface Rendering for Visualizing Large Datasets. *Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS*, 2001.
14. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro '95*.
15. R. V. Boppana and S. Chalasani. Fault-Tolerant Wormhole Routing Algorithms for Mesh Networks. *IEEE Transactions on Computers*, pages 848–864, July 1995.
16. S. Herrmann M. Castagnoli, G. Brauer. Optimization of cyclic redundancy-check codes with 24 and 32 paritybits. In *IEEE Transactions on Communication*, 1993.
17. U. Catalyurek, M. D. Beynon, C. Chang, T. Kurc, A. Sussman, and J. Saltz. The Virtual Microscope. *IEEE Transactions on Information Technology in Biomedicine*, 2002. To appear.
18. A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-aware Redirection. In *USENIX '99*.
19. Chelsio Communications. http://www.chelsio.com.
20. P. Culley, U. Elzur, R. Recio, and S. Bailey. Marker PDU Aligned Framing for TCP Specification, November 2002.
21. D. Dalessandro, A. Devulapalli, and P. Wyckoff. Design and Implementation of the iWARP Protocol in Software. In *PDCS '05*.
22. J. Gao and H. Shen. Parallel view dependent isosurface extraction using multi-pass occlusion culling. In *ACM SIGGRAPH*, 2001.
23. NetEffect Inc. http://www.neteffect.com/product-features.html.
24. InfiniBand Trade Association. http://www.infinibandta.org/.
25. Intel Core 2 Extreme quad-core processor. http://www.intel.com/products/processor/core2xe/qc_prod_brief.pdf.
26. H. M. Khosravi and A. Foong. Performance Analysis of iSCSI and Effect of CRC Computation. In *BEACON '04*.
27. Myricom. Myrinet home page. http://www.myri.com/.
28. F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects*, 2001.
29. G. F. Pfister and V. A. Norton. Hot-spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, 34:943–948, 1985.
30. Qlogic Corporation. http://www.qlogic.com.
31. R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt, April 2005.
32. D. V. Sarvate. Computation of cyclic redundancy checks via table look-up. In *Communications of the ACM*, volume 31, 1998.
33. A. Vishnu, M. Koop, A. Moody, A. Mamidala, S. Narravula, and D. K. Panda. Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective. In *CCGrid*, 2007.