USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance

Muralidharan Rangarajan* and Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019
{muralir,iftode}@cs.rutgers.edu

Abstract

In this paper, we describe an implementation of software Distributed Shared Memory (DSM) over Virtual Interface Architecture (VIA) for a Linux-based cluster of PCs and evaluate its performance. VIA is a user-level memory-mapped communication model that provides zero-copy communication and low-overhead by excluding the operating system kernel from the communication path. To our best knowledge, our implementation is the first software DSM protocol on VIA.

The DSM protocol we have implemented on VIA is Home-based Lazy Release Consistency (HLRC) that previous studies have shown to exhibit good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart. The experimental results obtained on seven Splash-2 applications show that VIA can be successfully used to support software shared memory on clusters of PCs. The paper is accompanied by a source-code distribution of the software DSM protocol for Linux/VIA clusters.

1 Introduction

System Area Networks (SANs) have become an increasingly popular solution to build scalable computer clusters by providing low latency and high bandwidth communication. Traditional communication models were unable to fully exploit the raw performance of the networks due to the high overhead added by the software protocols.

Virtual Interface Architecture (VIA) [5] is a user-level memory-mapped communication model for SANs, that reduces communication overhead by excluding the operating system kernel from the communication path. VIA is an industrial standard inspired from previous research in user-level communication performed in universities [9, 11, 10, 25]. The basic idea in user-level communication is to factor out the operating system from the critical path of communication operations. To provide protected communication, two conditions must be satisfied. First, the kernel must grant the permission for a process to communicate with another process by providing a communication channel. Second, the network interface must multiplex user-level DMA performed through these channels. This support eliminates the need to trap into the kernel each time a send is executed, and makes the send operation low-overhead. At the same time, by sending data from user space to a remote receive buffer, no copy is necessary and the end-to-end communication bandwidth will be close to the raw bandwidth provided by the network hardware.

There are multiple hardware and software implementations of VIA today. Giganet[12] has a hardware VIA implementation with drivers for Linux and Windows-NT. Firmware implementations of VIA are available for ServerNet[27] and Myrinet[22] interconnects. M-VIA [23] provides Linux software VIA drivers for various fast Ethernet cards.

The efficiency of memory-mapped communication provided by VIA doesn't come for free. As various projects started to use VIA or other memory-mapped communication libraries, it became obvious that the lack of buffer management, flow control and message packaging can make communication programming more complicated. The solution is to build high-level communication abstractions on top of VIA, while preserving its performance bene-

*Supported in part by a Student Research Grant from USENIX

fits. Recently, several message passing libraries over VIA, such as MPI [24], have been announced.

In this paper, we describe an implementation of software distributed shared memory (DSM) over VIA, for a Linux-based cluster of PCs. Software DSM [17] is available to applications as a runtime library that provides the abstraction of a shared address space across the cluster using message passing and virtual memory page protection. Given its low latency and overhead, as well as its capability to DMA directly into user address space of remote memory without intermediate copies, VIA appears very promising for software DSM. To our best knowledge, ours is the first implementation of a software DSM protocol on VIA.

The protocol we have implemented on VIA is home-based lazy release consistency (HLRC) [35, 17]. Previous studies have shown that HLRC provides good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart [35]. Home-based protocols have been previously implemented on other memory-mapped interconnected clusters both for clusters of uniprocessors [20, 15] as well as for clusters of symmetric multiprocessors (SMPs) [29, 26]. Although the communication model of these networks are similar to VIA, there are a number of significant differences. For instance, compared to the virtual memory-mapped communication (VMMC) implementation on Myrinet [9], VIA requires memory registration both for send and receive, has receive queues that can be combined into completion queues (on which threads can block on explicit receive). Compared to Memory Channel [13] used in [29], VIA has no broadcast support and no implicit global ordering.

Our goal is to implement a highly efficient home-based DSM protocol exploiting the features of the VIA model and investigate its overall performance as well as the performance impact of various VIA features. For the performance evaluation we used a set of seven Splash-2 applications [33] and a cluster of eight PCs connected by Giganet VIA-based cLAN network and running Linux version 2.2.10. We were able to obtain a speedup of greater than 6 for five applications. The performance we obtained is comparable to those previously reported for home-based protocols on Myrinet/VMMC connected clusters. We have learned from our performance study that even though VIA lacks features desirable for software DSM systems, like scatter-gather and broadcast support, the VIA primitives are a good match for the requirements of the software DSM communication model.

2 Virtual Interface Architecture

The VI Architecture [5] is a user-level memory-mapped communication architecture that is designed to achieve low latency, high bandwidth across a cluster of computers. The VI architecture attempts to reduce the amount of software overhead imposed by traditional communication models, by avoiding the kernel involvement in each communication operation. In traditional models, the operating system multiplexes access to the hardware between communication endpoints and therefore all communication operations require a trap into the kernel.

Each consumer process (VI Consumer) is provided a directly accessible interface to the network hardware, called the Virtual Interface (VI). Each VI represents a communication endpoint and pairs of VIs can be connected to form communication channels for bidirectional point-to-point data transfer. Each VI has a pair of work queues, one for send and one for receive. VI Consumers send and receive messages by posting requests, in the form of descriptors, to these queues. These requests are asynchronously processed directly by network interface controller (VI Provider) and marked with a status value when completed. VI Consumers can then remove these descriptors from the queue and reuse them if necessary. Completion queues allow the VI Consumer to combine the descriptor completion events of multiple VIs into a single queue.

There are several key features of the VIA communication model:

- Direct Access to the Network Interface. This enables low latency communication which has been shown to improve DSM performance.
- Memory Registration. VIA requires that memory used for every data transfer request be registered. Any memory page registered with VIA is kept pinned to the same physical memory location until the memory is deregistered by the VI Consumer. The necessity of memory registration becomes an issue for software DSM when the shared address space is larger than the physical memory or when memory pressure

due to other applications makes it difficult to register the entire shared address space.

- Zero-Copy Protocols. With memory registration, the VI Provider can transfer data directly between the buffers of a VI Consumer and the network without copying any data to or from intermediate buffers. Zero-copy communication protocols help improve the performance of DSM systems but because it requires registration of the entire address space, it can be used only for small problem sizes.
- Protected Channel for Communication. The VI architecture requires that a VI be explicitly connected with another VI in order to transfer data between them. Communication using the VI channels established by the connection process eliminates the protection check by the operating system from the critical path of data transfer. This feature is not relevant to software DSM systems that typically assume no sharing of the cluster with other applications.

The VI architecture supports two types of data transfer models for communication. The *Send-Receive* model is similar to traditional message passing, which involves an explicit receive operation, and the recipient of a message has to specify the memory location where the data will be placed. The *Remote Direct Memory Access* (RDMA) model involves only the sender, and no receive operation is required. In this case, both the source and destination buffer are specified by the sender. The VIA specification defines two RDMA operations, RDMA Write and RDMA Read.

3 Software DSM

Software DSM is a runtime system that provides the shared address space abstraction across a message-passing based cluster of computers. The basic idea suggested by Kai Li [21], is to use the virtual memory page protection mechanism to implement an invalidation-based coherence protocol similar to directory-based cache coherence, but at page granularity and completely in software. Since the unit of coherence is a virtual memory page, false sharing occurs when multiple unrelated shared objects lie on the same page. To alleviate the message traffic that would be generated in the presence of false sharing, several relaxed consistency models have been proposed [16, 4, 19, 6, 18]. These consistency models define a memory model for programmers in which they agree to exclusively use explicit synchronization. Under this assumption, the coherence protocol can delay the invalidation messages until a synchronization operation is performed, thus reducing both the protocol messages as well as the extra communication that an early invalidation would have unnecessarily caused.

3.1 Lazy Release Consistency

The most frequently used consistency model in software DSM is Lazy Release Consistency (LRC) [19, 6], in which the invalidations are propagated at acquire time. *Acquire* and *release* are the two explicit synchronization operations required in release consistency model and correspond to lock acquire and lock release respectively. A *barrier* is a global synchronization operation, implemented as a release followed by an acquire. In LRC, the updates are detected in software by computing diffs between the dirty page and a snapshot of the clean copy of the page.

The protocol that we chose to implement on VIA is HLRC [35]. The HLRC protocol implements a multiple-writer scheme by selecting a home for each page, to which updates are sent. The basic idea is to compute diffs at the end of an interval to detect updates and to transfer the updates as diffs to their homes. As a result, the home copy is up-to-date and can be used to update other non-home copies on demand. This protocol has been shown to have very good scalability: the number of messages necessary to update all copies is linear in the number of nodes and the memory overhead is constant [35]. The home-based protocol has also been shown to suit well with user-level memory-mapped communication because pages can be fetched from homes with no copy and diffs can be applied directly on the home's copy [15].

In software DSM, the explicit synchronization operations (acquire, release and barrier) are implemented using message passing. Each lock has a home through which the current owner of the lock is found. Usually, a distributed queue is used to implement queuing for lock acquires. Barriers can be implemented with a linear number of messages using a barrier manager or hierarchically using a logarithmic number of messages. In release consistent software DSM, invalidations are propagated as a list of *write-notices* at synchronization time.

3.2 Basic Programming Model

Typically, software shared memory provides an incomplete shared memory programming model. The execution model is based on multiple threads (one or more on each node) that share static global data in read-only mode, and dynamically allocated data in read-write mode. The coherence applies to the latter exclusively. Static data is usually updated by the main thread before the other threads are spawned. Also, all global shared memory allocations must be performed by the main thread before the other threads are spawned. Since static data cannot be modified once the threads are spawned, it is typically used to maintain pointers to the shared data.

Applications written to use our DSM system make use of the parmacs macros, which were developed at ANL. The macros provide platform independence to the application, enabling it to run on software DSM as well as hardware DSM systems without modification. These macros provide a minimum set of primitives that are necessary in order to program a shared memory application.

The protocol implements the multi-threading model by "forking" one process on each node of the cluster. Each process will execute at least one application thread. The threads will share the address space within the process as well as across the forked processes using software DSM. Since Linux doesn't provide a remote fork, we provide the "illusion" of this by starting the same executable on each node using *rsh*. Each remote process executes the same code as the initial process did before spawning, to initialize static data, making it coherent across nodes.

4 Protocol Design

In this section, we explain the design of the HLRC protocol. We describe the entry points to the protocol by specifying for each entry point the protocol actions and the messages used to perform these actions.

4.1 Protocol Entry Points

Protocol activity occurs at various points in the execution of an application. The entry points to the protocol can be *synchronous* or *asynchronous*. *Synchronous* entry points are those at which the application traps into the protocol and executes some protocol action. The *asynchronous* entry points are entered as a result of incoming messages generated by protocol action on other nodes in the system.

4.1.1 Synchronous Entry Points

During its execution, the application can enter the protocol *synchronously* for the following:

- Lock Acquire - When the application needs a lock, it depends on the underlying HLRC protocol to get the lock from the current owner and perform the appropriate coherence actions.
- Lock Release - When the application needs to release the lock, it uses the HLRC protocol to manage the released lock and perform the appropriate coherence actions.
- Barrier - The application depends on the HLRC protocol to implement a barrier among the participating nodes.
- Page fault - When the application tries to access shared data which has been invalidated as a result of a coherence action, a page fault is generated. The page fault handler, installed by the HLRC protocol at initialization, will fetch the shared page from its home.

4.1.2 Asynchronous Entry Point

The synchronous entry points generate request messages which have to be serviced at the receiving node. The HLRC protocol provides an asynchronous entry point to process the received messages. This can be implemented in several ways:

- Hardware - If support is available in the network interface for asynchronous message handling (for instance with a complete implementation of the VIA specification, a page request can be serviced with an RDMA Read).
- Interrupt Handler - Interrupt handlers can be used to receive and process remote requests if notifications are issued on message arrival.

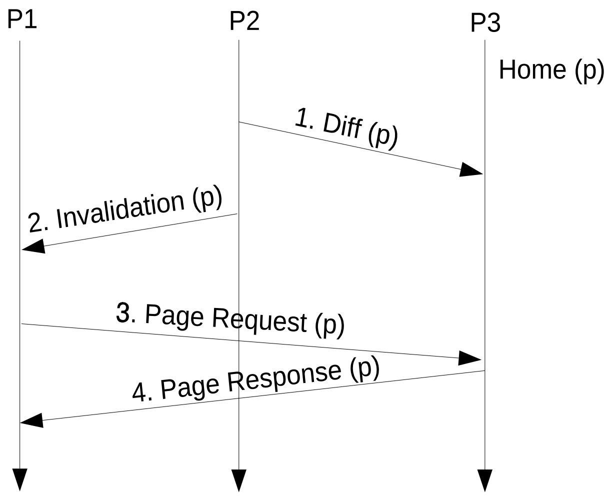

Figure 1: Coherence Messages exchanged by three processes in HLRC

- Communication Thread - A separate communication thread can be used to handle messages, using either polling or blocking.

We use Giganet's implementation of VIA, which does not support RDMA Read or asynchronous notification. Therefore, the asynchronous entry point in our implementation is covered by a separate communication thread on each node that is responsible for handling all the incoming messages.

4.2 Protocol Messages

The protocol activity generates two types of messages: *coherence messages* and *synchronization messages.*

The *coherence messages* are related to update propagation and fall in one of the following categories:

- Diffs - sent by a writer of the page to the home of the page at release or acquire time; contain the updates performed by the sender since the last release or acquire.
- Invalidations - sent at acquire time by the last releaser; contain a list of pages that were updated at the last releaser and elsewhere, that the acquirer must update.
- Page fetch request - sent at page fault time to the home.
- Page response - sent by the home to the faulting node as a response to the page fetch request message.

Figure 1 illustrates the flow of coherence messages when a shared page (p) is updated by a process (P2) and subsequently accessed by another process (P1). The timing and order of the coherence operations are determined by the consistency model implemented by the DSM system. For example, in homeless LRC, the diff messages are sent lazily on demand, while in the home-based LRC, diffs are sent eagerly, either at release or acquire time.

The *synchronization messages* are used to implement the distributed queue for locks and the distributed barrier. In most software DSM protocols, especially in LRC, coherence messages and synchronization messages are combined in a single message whenever possible. For example, in LRC, the invalidation message is combined with the reply message to a lock acquire.

4.3 Implementation of HLRC on VIA

4.3.1 Data Transfers

The data transfers (pages and diffs) are performed using RDMA Write with or without copy. If the problem size is small, the entire shared address space is registered with VIA, and page transfers from home to the non-home nodes are performed without any copy. Memory registration consists of locking the pages of a virtually contiguous memory region into physical memory and providing the virtual to physical translations to the NIC. The amount of physical memory on the machine imposes a limit on the amount of memory that can be registered. If the problem size is larger than the limit imposed by the VIA implementation for memory registration, a set of communication buffers are registered instead and page transfers are performed with one copy at each end (from page to buffer at the sender and from buffer to page at the receiver). Unlike VMMC, another memory-mapped communication library that requires receive buffers to be registered ("exported" in its terminology), VIA requires both send and receive buffers to be registered.

To send an update, the diff is computed by packing all the modified words within a dirty page into one message by the sender (non-home node), and sent to

Applications	Problem Size	Sequential Time (s)	Shared memory size
Barnes-Spatial	262144 bodies	357	325 MB
FFT	2048x2048	86	196 MB
LU	2048 x 2048	209	33 MB
Ocean	514 x 514	30	97 MB
Radix	45M keys	95	377 MB
Water-Nsquared	32768 molecules, 5 steps	22450	22 MB
Water-Spatial	262144 molecules	14202	264 MB

Table 1: Application characteristics

the home of the page. The receiving node applies the diff by modifying the appropriate page at the words mentioned in the diff message.

4.3.2 Remote Requests

The DSM protocol may issue remote requests for data and synchronization. These requests, *which require a response*, are sent using the send-receive model. Since each node executes one application thread, there can be only one outstanding request issued by that node and, one corresponding reply. Therefore, each node expects at most N-1 requests (one from each other node). This means that each node must register N-1 receive buffers and post the same number of receive descriptors, where N is the number of nodes in the cluster. A N-th registered receive buffer is used to receive the reply messages (acks, locks, etc). Since VIA does not support notification on message arrival, a server thread is run on each node, which is responsible to handle remote requests. When no requests are pending, the server thread blocks on a completion queue that aggregates the receive queues for the N-1 buffers on which the node can receive asynchronous requests.

Messages *that do not require a response* (barrier, reply messages) are sent using RDMA Write and do not consume a descriptor on the receiving side. These messages are consumed in a busy loop by the application (not server) thread, since there is nothing else the application thread can do. The memory location for the flag on which spinning is performed, is updated by RDMA Write.

5 Performance Evaluation

5.1 Applications

We evaluated the performance of our DSM system using seven applications from the SPLASH-2 benchmark suite [33]: Barnes, FFT, LU decomposition, Ocean, Radix, Water-Spatial and Water-Nsquared. Due to space limitations, we don't describe the applications in our paper. In Table 1, we show the problem size, sequential execution time and the shared memory footprint for each of these applications.

5.2 Experimental Platform

All our experiments were performed on a cluster of eight SMP PCs. Each PC contains two 300 MHz Pentium II processors. However, for this study, we used only one processor on each node. Each processor has a 512KB L2 cache and each node contains 512 MB of main memory. All nodes run Linux-2.2.10.

Each node has a Giganet cLAN NIC, which is a 32-bit 33 MHz PCI-based card. These nodes are connected by an 8-port Giganet cLAN switch. The performance characteristics for our experimental platform are reported in Table 2. Latency denotes the time taken to transfer a 1 word packet between two nodes using VIA. PostSend denotes the average time taken to post a send using VIA. The last row presents the cost of the VipRegisterMem operation used to register memory used for communication buffers in VIA.

We also present (Table 3) the cost of other operations or events that occur frequently in a software DSM system: page fault handler invocation, the mprotect system call, and memory copy bandwidth.

One-way Latency (1 word)	8.2 μs
Bandwidth (32 KB)	101 MB/s
PostSend (4 KB)	2.1 μs
RegisterMem (4 KB)	4.3 μs

Table 2: Giganet VIA Microbenchmarks

The last row in Table 3 presents the time taken to copy a page(4096 bytes on the Pentium II running Linux) from memory to cache.

Operation (per page)	Time (μs)
Page fault	6.2
Mprotect call	2.7
Memory copy	23.2

Table 3: Linux System Microbenchmarks

In Table 4, we present some microbenchmarks for the DSM system itself. To derive the basic cost of all these operations, these microbenchmarks were done using just two nodes. The Acquire microbenchmark gives the time to update data structures and fetch the lock from a remote node. The Release microbenchmark measures the cost of a release without any pending request for the lock. The page fetch time indicates the time to fetch a page from home without copies. The diff application time includes the time to copy the diff from the diff buffer onto the page and update the version of the page. The Barrier microbenchmark includes the time to send the barrier message to the other node, and wait for the barrier message from the other node.

Operation	Time (μs)
Acquire (Local, Remote)	1, 34
Release	1
Page fetch (no copy)	89
Diff Computation	24
Diff Application	22
Barrier(2-node)	17

Table 4: Software DSM Microbenchmarks

5.3 Application Performance

We ran the seven applications on our cluster of eight nodes. On each node, the application consists of two threads, the communication thread for handling incoming messages and the application thread that performs computation. We present the performance results for the problem sizes mentioned in Table 1 and then analyze the performance in detail.

Table 5 shows the speedups for the seven SPLASH-2 applications we used. LU and Ocean achieved speedups of 7.4 and 7.7 respectively, followed by Water-Spatial, Barnes and Water-Nsquared with speedups greater than 6. FFT comes next followed by Radix which has the worst speedup of the lot.

Applications	Speedup (8 nodes)
Barnes	6.3
FFT	5.8
LU	7.4
Ocean	7.7
Radix	4.3
Water-Nsquared	6.2
Water-Spatial	6.7

Table 5: Speedups on 8 nodes

For the purpose of this study, we classify the applications according to their data access patterns and synchronization behavior. The application can be *single writer* or *multiple writer*, based on the number of concurrent writers on the same coherence unit (a page). The communication to computation ratio is determined by the granularity of data access. *Fine grain* access can introduce fragmentation and/or false sharing, resulting in an increase in the communication to computation ratio. Since all coherence events in the LRC protocols happen at synchronization points, the *frequency of synchronization* plays an important role in the performance. The average computation time between two consecutive synchronization events is a good measure of the frequency of synchronization.

LU and Ocean are single-writer applications with coarse-grain access. These applications exhibit good spatial locality with only one writer per shared page and hence achieve good speedups. FFT is a single-writer application with fine-grained access. The mismatch between the access granularity and the communication granularity prevents it from achieving a better speedup. Applications like Barnes-Spatial and Water-Spatial are multiple-writer with fine-grain access and coarse-grain synchronization. The high average time between synchronization events for these applications helps in achieving good performance. The relaxed consistency model and the multiple-writer support of HLRC helps these applications in achieving good speedups. Water-Nsquared and Radix are multiple-writer applica-

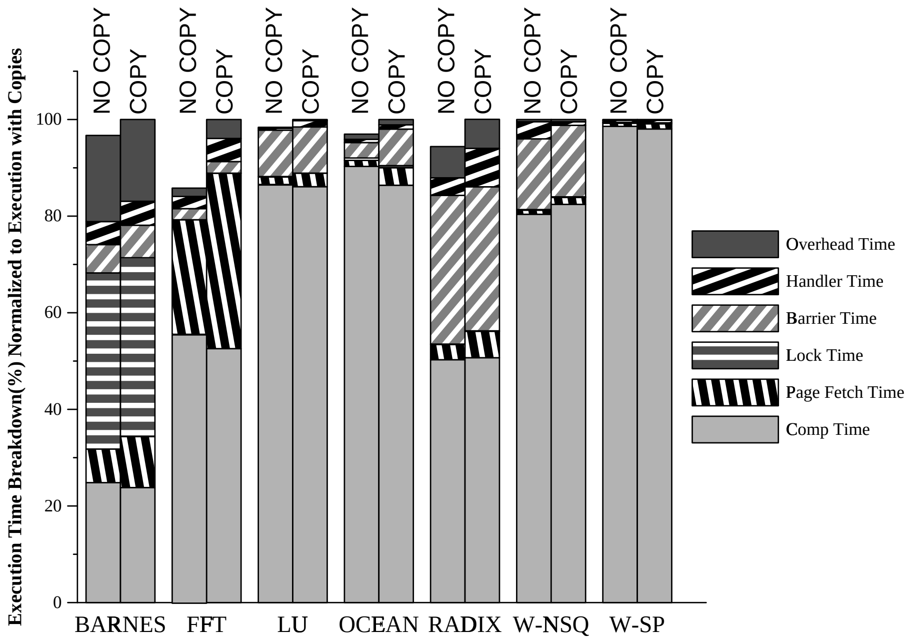

Figure 2: Normalized execution time breakdown on 8 nodes

tions with coarse-grain access. In Water-Nsquared, since each process updates successively a large number of contiguous molecules, the access pattern is preserved at the page level which leads to a coarse-grain access pattern, which is well suited. Radix, however, does not achieve a good speedup due to a large amount of time spent in the barrier, which is caused by an imbalance.

5.4 Performance Impact of Copies

We evaluate the impact of copies, necessary as part of data transfer when the entire shared address space cannot be registered with VIA, by presenting a comparison between the performance of the no-copy and copy versions in Figure 2. We present a comparison of the execution times breakdown for both versions, normalized with respect to the executions with copies. We had to run the applications with problem sizes smaller than the ones mentioned in Table 1 so that we could use both versions with the same problem size. The bars on the left, labeled "NO COPY", present the performance results for the no-copy version, and the bars on the right, labeled "COPY", present the performance results for the version with copies. Each bar presents a percentage breakdown of the different components which make up the execution time on a single node. Computation time is the time spent doing application computation. Page fetch time is the time spent in fetching a page from the home node, on a page miss. Lock time is the time spent in getting the lock from the current owner. Barrier time is the time spent waiting for barrier messages from other nodes, at the barrier. Overhead time is the time spent performing protocol actions. Handler time is the time spent inside the handler, servicing remote requests. Since we used only one processor on each node, for our experiments, the handler competes for the CPU with the application thread to service the messages received via the receive completion queue.

The page fetch time is what increases as a result of the additional copies at the home node and the receiving node during page transfers. We can see that *Page Time* makes up for a significant percentage of the execution time for Barnes, FFT and Radix, and these three applications show an improvement in performance with copy avoidance. Although avoiding copy is good, data transfer with copies doesn't degrade performance drastically. The performance degradation was maximum for FFT (15%) and very little (less than 5%) for the other applications.

6 Discussion

In this section, we present the lessons we learned from this implementation. In particular, we discuss the potential and limitations of the current VIA specification and implementations, for software DSM.

Low-latency Communication. VIA provides low latency communication which is critical for the performance of a DSM system. Figure 3 presents the percentage distribution of the message sizes for four of the applications. For all four applications, small messages (less than 256 bytes in size) constitute more than 75% of the total number of messages.

Copy Avoidance. Copies can be avoided in data transfers but VIA requires both the send and receive buffers to be registered in advance. The cost of memory registration (Table 2) prevents us from doing it at the time of transfer. On the other hand, any VIA implementation imposes a limit on the amount of memory that can be registered. As a result, for large problem sizes, copies cannot be avoided. However, from the results presented in Section 5, we can see that performing copies as part of data transfer doesn't adversely affect application performance except in the case of FFT, where we observed a degradation of roughly 15%.

Scatter-Gather. A scatter-gather mechanism would have been ideal to implement direct diffs without incurring the penalty of multiple message latencies. In the absence of scatter-gather, preliminary calculations indicate that direct diff solutions win over the diff copy solution only when the chunks of consecutive updates are large enough to offset the latency of sending multiple messages using VIA.

To understand the impact of writing diffs directly, avoiding copies but without scatter-gather, we looked at two of the applications, viz., Radix and Barnes which generate a substantial amount of diff traffic. When diffs are written directly, a message is generated for every contiguous dirty segment in the page. Radix achieves an improvement in performance by writing diffs directly, whereas the performance of Barnes degrades. On a careful look at the granularity of the writes and the number of dirty segments per modified page, we realized that Radix resulted in only one contiguous dirty segment per page, whereas Barnes resulted in about 21 dirty segments per page. For Barnes, the overhead of sending multiple dirty segments per page outweighs the improvement achieved by avoiding the copy.

What VIA provides as scatter-gather support is however insufficient for the implementation of direct diffs with one message per page. VIA allows the source of an RDMA Write to be specified as a list of *gather* buffers. However, this gather mechanism doesn't allow us to specify multiple addresses on the destination node. In software DSM, transfer of diffs for any page involves transfer of multiple contiguous dirty segments contained within the page.

We try to estimate the potential performance improvement with scatter-gather support from VIA. We can calculate this by subtracting the time to apply the diff from the handler time. Knowing the total diff size that was transferred and approximating the diff application time with the memory copy time, for all seven applications we studied, we got a gain of no more than 5%. This is consistent with what other people have shown [2].

Remote Read. RDMA Read is a VIA feature that allows fetching of data without interrupting the processor on the remote node. Although present in the VIA specification, the VIA implementation that we used in our experiments does not support RDMA Read. We try to make a rough approximation of the impact of RDMA Read on the performance results.

Using RDMA Read, we can potentially eliminate the handling time for remote requests (since they can be performed by the NIC as an RDMA Read), assuming that RDMA Reads do not require servicing by the CPU. Even though not all remote requests are remote fetches, we look at an upper bound by assuming that the entire handling time is eliminated. For all the applications that we studied, this component (handling incoming messages as a server) of the execution time is not larger than 5%. The elimination of the remote handling time, would also reduce the communication latency experienced by the clients, by the same amount. This brings the total contribution of the remote read to no more than 10%, not counting the side-effect on synchronization due to critical section dilation [2]. Bilas et al [2] have shown that the remote read facility can help reduce the page fetch times by about 20% for most applications.

Broadcast Support. VIA doesn't specify any primitive or mechanism for broadcast. Broadcast can be really useful in the context of a software DSM system. With support for inexpensive broadcast, we can adopt an eager selective update mechanism using broadcast, instead of sending write notices for invalidation. This will help us save unnecessary page requests generated at nodes accessing heavily accessed pages, and in reducing the contention and protocol overhead of serving these pages at the home nodes. We can also broadcast the invalida-

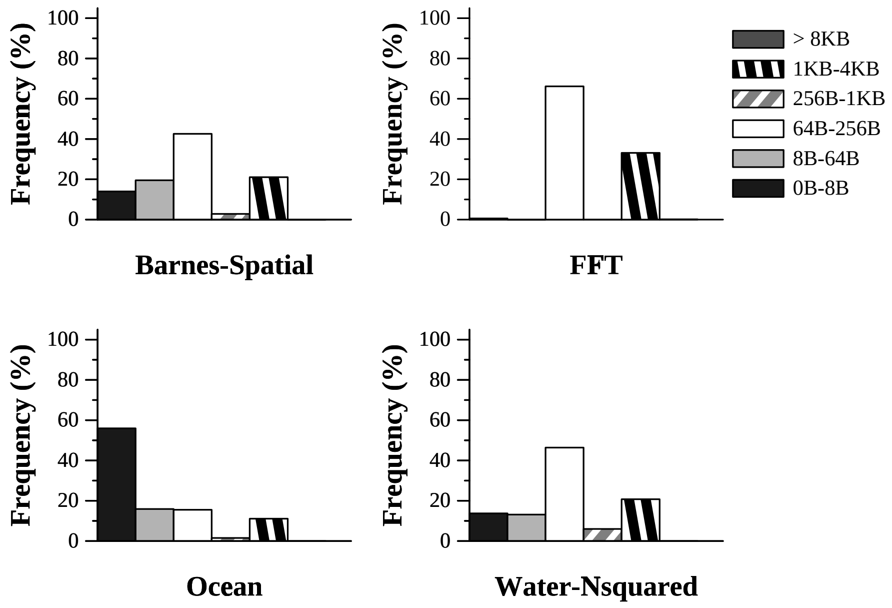

Figure 3: Message size distribution

tions sent at the time of barriers. Previous studies [30] have revealed that a gain of up to 13% could be achieved over 8 nodes, with selective use of broadcast for data used by multiple consumers. They present simulation studies to speculate that a performance improvement of even 50% is possible with 32 nodes.

7 Related Work

This work focuses on using memory-mapped communication to build a high-performance software DSM. In this context, we evaluate VIA as an effective communication substrate for software DSM.

A great deal of work has been done on shared virtual memory since it was first proposed[21]. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence. RC was used to reduce false sharing by allowing multiple writers [4]. Lazy Release Consistency (LRC) [19, 6] further relaxed the RC protocol to reduce protocol overhead. Treadmarks [18] was the first SVM implementation using the LRC protocol on a network of stock computers. The Automatic Update Release Consistency (AURC) [14] protocol was the first proposal to take advantage of memory-mapped communication to implement an LRC protocol. Home-based Lazy Release Consistency (HLRC) [17] proposed a home-based approach to improve the performance on large-scale machines. Cashmere [20] is an eager Release Consistent (RC) SVM protocol that implements a home-based multiple-writer scheme using the I/O remote write operations supported by the DEC Memory Channel network interface [13].

The VI architecture [5] builds on previous work in user-level communication. The VI architecture is based on ideas similar to that of U-Net [11], virtual interfaces to the network from application device channels [7], and Virtual Memory Mapped Communication (VMMC) [8]. Other research that discuss user-level direct access to the network interface are FM [25], AM [10], Hamlyn [32], PM [31], and Trapeze [34].

Prototype implementations of the VI Architecture have been developed on Myrinet, and 100 Mb/s Ethernet. M-VIA [23] is a software emulation of VIA over various network interface cards including Ethernet cards. Berkeley VIA [3] is an implementation of VIA over Myrinet. A performance study of VIA [28] has compared software as well as hardware implementations. The study also explores several performance and implementation issues related to the use of VIA by distributed applications.

Previous work [2, 30, 1] has looked at exploiting support available in hardware to improve the performance of software DSM. Bilas et al [2] explore performance gains to be obtained from performing asynchronous message handling in the network interface. Another study [30] investigates the impact of features such as low-latency messages, protected remote memory writes, inexpensive broadcast and total ordering of network packets on the performance of software DSM. The use of a PCI-based programmable protocol controller for hiding coherence and communication overheads in software DSMs, is studied in [1].

This work sets out to illustrate the match between software DSM requirements and the memory-mapped communication features offered by VIA. To our knowledge, ours is the first performance study of software DSM over VIA.

8 Conclusions

We have implemented a high-performance software distributed shared memory protocol for clusters of PCs connected by Virtual Interface Architecture networks. In this paper, we describe the implementation of a Home-based Lazy Release Consistency DSM protocol on VIA and evaluate its performance on a eight node cluster of PCs using 7 benchmark applications from the Splash-2 suite.

We observe that the VIA primitives are a good match for the requirements of the software DSM communication model. We have learned from our performance study that desirable features for software DSM systems, like scatter-gather, broadcast support, are missing from VIA. Even though the memory registration mechanism imposes a limit on the problem size that can be handled with a zero-copy protocol, our performance studies reveal that copies do not affect the application performance adversely.

The experimental results show that VIA can be successfully used to support shared memory on clusters of PCs but further study is necessary to evaluate its scalability on larger clusters and for a larger set of applications.

Acknowledgments

We thank the referees for their helpful suggestions and our shepherd, Matt Welsh, for working with us to improve the paper. We thank Thu Nguyen, Ricardo Bianchini and Florin Sultan for helpful discussions, and Zoltan Jarai for help in formatting this document. This work was supported by the Rutgers Sciences Council Projects Program, USENIX Research Grant, Microsoft, and NSF CISE-9986046.

Availability

A software distribution package with the software DSM protocol described in this paper is free, and available for download from
`http://discolab.rutgers.edu/projects/dsm`

References

[1] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[2] Angelos Bilas, Cheng Liao, and Jaswinder Pal Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[3] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SuperComputing Conference*, November 1998.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0. http://www.viarch.org*, 1997.

[6] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] Peter Druschel, Bruce S. Davie, and Larry L. Peterson. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the ACM SIGCOMM Symposium*, September 1994.

[8] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[9] Cezary Dubnicki, Angelos Bilas, Kai Li, and Jim F. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[10] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[11] 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 Symposium on Operating Systems Principles*, pages 40–53, 1995.

[12] GigaNet. *http://www.giganet.com*.

[13] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects Symposium*, August 1995.

[14] L. Iftode, M. Blumrich, C. Dubnicki, D.L. Oppenheimer, J.P. Singh, and K. Li. Shared Virtual Memory with Automatic Update Support. In *Proceedings of the 13th ACM International Conference on SuperComputing*, June 1999.

[15] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[16] L. Iftode and J.P. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of the IEEE*, 87(3):498–507, March 1999.

[17] Liviu Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, 1998.

[18] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[19] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[20] L.I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[21] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[22] Myricom. *http://www.myri.com*.

[23] National Energy Research Scientific Computing Center. *M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via*, 1999.

[24] National Energy Research Scientific Computing Center. *MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html*, 1999.

[25] Scott Pakin, Mario Laura, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of SuperComputing Conference*, 1995.

[26] R. Samanta, A. Bilas, L. Iftode, and J.P Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, 1998.

[27] ServerNet. *http://www.servernet.com*.

[28] W. E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the Virtual Interface Architecture. In *Proceedings of the 13th ACM International Conference on Supercomputing (ICS)*, June 1999.

[29] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L.I. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[30] R. Stets, S. Dwarkadas, L.I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order and Remote-Write Capability on Network-based Shared Memory Computing. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, 2000.

[31] H. Tezula. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of the International Parallel Processing Symposium*, pages 308–314, 1998.

[32] John Wilkes. Hamlyn – An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1993.

[33] 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 23rd Annual Symposium on Computer Architecture*, May 1995.

[34] K. Yocum. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 243–252, 1997.

[35] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.