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 a 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 SAN’s 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) [6] is a user-level memory-mapped communication model for SAN’s that further reduces the 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 [10, 12, 11, 25]. The basic idea in user-level communication is to factor out protection from the critical path of the 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 the 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 has a hardware VIA implementation with drivers for Linux and Windows-NT. Firmware implementation of VIA are available for ServerNet (Tandem) and Myrinet (Myricom) 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 have started to use VIA or other memory-mapped communication libraries,
it became obvious that the 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 benefit. Recently, several message passing libraries over VIA(sockets, MPI) have been announced[24].

In this paper, we describe an implementation of a software distributed shared memory over VIA for a Linux-based cluster of PCs. Software DSM [18] 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. There are two issues related to the performance of page-based software DSMs. First, the unit of coherence is a virtual memory page which is typically larger than the granularity of sharing in the application. This leads to false sharing which can generate unnecessary communication. To address this, many relaxed consistency models and lazy protocols have been proposed [5, 20, 7, 19]. Second, the performance of software DSM critically depends on message latency. 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 to offer adequate communication support for software DSM. To our best knowledge, our implementation is the first software DSM protocol on VIA.

The protocol we have implemented on VIA is home-based lazy release consistency (HLRC) [33, 18]. Previous studies have shown that HLRC provides good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart [33]. Home-based protocols have been previously implemented on other memory-mapped interconnected clusters both for cluster of uniprocessors [21, 16] as well as for cluster of symmetric multiprocessors (SMPs) [28, 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 [10], VIA has no notification, requires memory registration both for send and receive, has receive queues that can be arbitrarily combined into completion queues (on which threads can block on explicit receive). Compared to Memory Channel [14] used in [28], 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 [32] and a cluster of eight PCs connected by GigaNet VIA-based C-LAN network and running Linux version 2.2.10. Except for two applications(out of the seven we studied), we were able to obtain a speedup of greater than 6. The performance we obtained is comparable to those previously reported for home-based protocols on Myrinet/VMMC connected clusters.

The paper is accompanied by a source-code distribution of the software DSM protocol for Linux/VIA clusters. Although many optimizations have been proposed to improve the home-based protocols, we deliberately choose to keep the first release simple. We believe that this will allow the user community to understand it quickly and to investigate other protocol design tradeoffs for VIA and Linux.

2 Virtual Interface Architecture

The VI Architecture [6] 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.

The VI architecture is composed of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers.

The VI Consumer represents an application. Each consumer process is provided a directly accessible interface to the network hardware, called the Virtual Interface (VI). Each VI represents a
communication endpoint and pairs of such VIs can be connected to form communication channels for bidirectional point-to-point data transfer.

The VI Provider consists of hardware and software components. The Network Interface Controller (NIC) makes up the hardware component and a kernel agent, the software component. The kernel agent is usually a driver supplied by the NIC vendor, and is responsible for the resource management functions needed to maintain a VI between VI Consumers and VI NICs.

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 processed by the VI Provider asynchronously and marked with a status value when completed. VI Consumers can then remove these descriptors from the queue and reuse them if necessary. A doorbell mechanism, which is directly implemented in the VI network adapter, is used to notify the network adapter that a new descriptor (request) has been posted to a Work Queue. Completion queues allow the VI Consumer to combine the descriptor completion events of multiple VIs into a single queue.

2.1 Direct Access to the Network Interface

The VI architecture provides applications with direct access to the network interface. The direct access to the network interface facilitates transfer of data directly from the address space of the user to the destination address space on a remote machine. In short, this enables Remote DMA from user space. The application specifies the address of a memory location as the source of a send and posts a send request. The doorbell mechanism is then used to directly access the network interface adapter for this request. The VI provider takes the specified source address from the posted request and performs the transfer for the user.

2.2 Memory Registration

Most traditional systems require memory pages used in data transfers to be pinned to physical address locations before being accessed by the NIC. VIA requires the VI Consumer to register the memory used for every data transfer request. Any memory page registered with VIA is kept pinned to the same physical memory location until the memory is deregistered by the VI Consumer. By means of the one-time memory registration, the cost of pinning memory pages can be eliminated from the critical path of data transfer.

2.3 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 performance by avoiding data copies which form a large component of communication overhead and consume memory bandwidth.

2.4 Protected Channel for Communication

The VI architecture provides connection-oriented data transfer service. A VI must be explicitly connected with another VI in order to transfer data. Once connected, these VIs represent protected channels for bidirectional transfer of data. Communication using the protected VI channels eliminates the protection check from the critical path of data transfer.

2.5 Bypassing the Kernel

With the VI architecture, each VI represents a communication endpoint. It is the network adapter which performs the endpoint virtualization directly which includes multiplexing, de-multiplexing, and data transfer scheduling. Since each consumer process is provided with a protected, directly accessible interface to the network, a VI Consumer can bypass the kernel for all data transfer operations, eliminating the overheads associated with these tasks.

2.6 Communication Models

The VI architecture supports two types of data transfer models for communication:

- Send-Receive model: This 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.

- Remote Direct Memory Access (RDMA): In this model, both the source and destination buffer are specified by the initiator of the data transfer. The data transfer involves only the initiator, and no receive operation is required. The VIA specification defines two RDMA operations, RDMA Write and RDMA Read. However, current VIA implementations support only RDMA Write.

3 Software Distributed Shared Memory

Software distributed shared memory (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 [22] 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 [17, 5, 20, 7, 19]. 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.

The most frequently used consistency model in software DSM is Lazy Release Consistency (LRC) [20, 7] in which the invalidations are propagated at the 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.

To reduce even further the extra traffic generated by false sharing, a LRC software DSM protocol usually support a multiple-writer coherence scheme based on diffs. With such a scheme, ownership is not required for writing, and in the presence of false sharing, multiple writable copies of the same page are allowed. Writes are collected and merged by computing the differences (diffs) between each dirty copy of the page and a clean copy created before the first write following an invalidation.

The protocol that we chose to implement on VIA is Home-based Lazy Release Consistency (HLRC) [33]. An HLRC protocol implements a multiple-writer scheme by selecting a home for each page to which diffs are sent. As a result, the home copy is up-to-date and can be used to update the other non-home copies on demand. This protocol has been shown to have a 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 [33]. The home-based protocol has been also 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 [16].

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 Consistency software DSM, invalidations are propagated as a list of write notices at synchronization time.

4 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 the static global data in read-only mode, and the dynamically allocated data in read-write mode. The coherence applies to the latter ex-
clusively. 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.

The software shared memory runtime library is provided through the parma package. The application uses the software shared memory library through an API which is specified in terms of macros defined in the parma package, developed at ANL. The parma package is implemented via macros using the M4 macro preprocessor which is available on most UNIX systems. The macros provide platform independence to the application, enabling it to run on software DSM as well as hardware DSM systems without any modification. These macros provide a minimum set of primitives that are necessary in order to program a shared memory application. Table 1 contains a list of macros which are part of the API used by applications.

A simple shared memory program is given in Figure 1.

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.

5 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.

5.1 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.

During its execution, the application can enter the protocol synchronously in the following cases:

- 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 release 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.

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:

Figure 2: Coherence Messages exchanged by three processes in HLRC

![Coherence Messages Diagram](image-url)
```c
#include <stdio.h>

// Create the environment to use the HLRC protocol
MAIN_ENV

// Number of nodes to be used for the computation
#define NUMPROC 8

// Address of shared variable
int *shared_counter;

main(int argc, char *argv[]) {
  int i;

  // Initialize the environment for the distributed computation
  MAIN_INITENV()

  // Allocate memory from the shared memory pool
  shared_counter = (int *)G_MALLOC(sizeof(int))

  // Initialize lock variable
  LOCKINIT(testlock)

  // Spawn processes on a number of nodes for the distributed
  // computation. The slaves terminate after the computation and
  // do not return to the main program. The master executes the
  // work function outside the loop and continues to the end.
  for (i = 1; i < NUMPROC; i++) {
    CREATE(work_func)
  }
  work_func();

  // Wait for computation to finish on other nodes
  WAIT_FOR_END()

  // Terminate the master process
  MAIN_END()
}

// The function which is to be invoked in parallel
void work_func() {
  int my_id, my_count;

  // Get the ID for this node
  GET_PID(my_id)

  LOCK(testlock)

  // Increment and get snapshot of shared counter
  my_count = (*shared_counter)++;

  UNLOCK(testlock)
}
```

Figure 1: A simple shared memory program using the DSM protocol
<table>
<thead>
<tr>
<th>Macro</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN_ENV,EXTERN_ENV</td>
<td>Defines environment</td>
</tr>
<tr>
<td>MAIN_INITENV</td>
<td>DMS initialization</td>
</tr>
<tr>
<td>MAIN_END</td>
<td>Terminate application</td>
</tr>
<tr>
<td>G_MALLOC</td>
<td>Allocate global shared memory</td>
</tr>
<tr>
<td>G_FREE</td>
<td>Deallocate global shared memory</td>
</tr>
<tr>
<td>CREATE</td>
<td>Spawn off thread to perform work</td>
</tr>
<tr>
<td>GET_PID</td>
<td>Get the ID assigned to this thread by the DSM system</td>
</tr>
<tr>
<td>LOCKINIT</td>
<td>Initialize lock</td>
</tr>
<tr>
<td>LOCK</td>
<td>Acquire lock</td>
</tr>
<tr>
<td>ULOCK</td>
<td>Release lock</td>
</tr>
<tr>
<td>BARINIT</td>
<td>Initialize barrier</td>
</tr>
<tr>
<td>BARRIER</td>
<td>Invoke barrier</td>
</tr>
<tr>
<td>WAIT_FOR_END</td>
<td>Master waits for slave threads to finish</td>
</tr>
</tbody>
</table>

Table 1: Macros used by applications

- **Diffs** - sent by a writer of the page to the home of the page at the 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 the 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 2 illustrates the flow of the 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.

### 5.2 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 asynchronously received messages. The asynchronous entry point 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.

- **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.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Problem Size</th>
<th>Sequential Time (s)</th>
<th>Shared memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes-Spatial</td>
<td>32768 bodies</td>
<td>38.91</td>
<td>40 MB</td>
</tr>
<tr>
<td>FFT</td>
<td>1024x1024</td>
<td>17</td>
<td>49 MB</td>
</tr>
<tr>
<td>LU</td>
<td>2048 x 2048</td>
<td>209</td>
<td>33 MB</td>
</tr>
<tr>
<td>Ocean</td>
<td>514 x 514</td>
<td>30</td>
<td>97 MB</td>
</tr>
<tr>
<td>Radix</td>
<td>8M keys</td>
<td>17</td>
<td>66 MB</td>
</tr>
<tr>
<td>Water-Nsquared</td>
<td>4096 molecules, 3 steps</td>
<td>363</td>
<td>28 MB</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>32768 molecules</td>
<td>2434</td>
<td>33 MB</td>
</tr>
</tbody>
</table>

Table 2: Application characteristics

6 Performance Evaluation

6.1 Applications

We evaluated the performance of our DSM system using seven applications from the SPLASH-2 benchmark suite[32]: 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 2, we show the problem size, sequential execution time and the shared memory footprint for each of these applications.

6.2 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 3. Latency denotes the time taken to transfer a zero-byte packet between two nodes using VIA. Bandwidth represents the rate at which data can be transferred between two processes. Finally, the send overhead denotes the average time taken to post a send request using VIA.

We also present 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. Page faults are generated on access of a previously protected page. Table 4 presents the invocation time for the page fault handler. The mprotect system call is used to change the protection attributes of a virtual memory page. In a software DSM system, mprotect is used to achieve invalidation and subsequent validation of shared pages. Table 4 contains the cost of the mprotect system call in Linux. The memory copy latency is the time taken to copy a segment in memory. The last row in Table 4 presents the time taken to copy a page (4096 bytes on the Pentium II running Linux) from memory to memory.

6.3 Application Performance

We ran the seven applications on our cluster of eight nodes. We first present the overall application performance results and then analyze the performance in detail. First, we evaluate a version of the protocol in which the entire shared memory used by the application is registered with VIA, thus avoiding copies on both send and receive sides. On each node, the application consists of two threads, the communication thread for handling incoming messages and the application thread that performs computation.
<table>
<thead>
<tr>
<th>Applications</th>
<th>Speedup (8 nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>6.3</td>
</tr>
<tr>
<td>FFT</td>
<td>5.8</td>
</tr>
<tr>
<td>LU</td>
<td>7.4</td>
</tr>
<tr>
<td>Ocean</td>
<td>7.7</td>
</tr>
<tr>
<td>Radix</td>
<td>4.3</td>
</tr>
<tr>
<td>Water-Nsquared</td>
<td>6.0</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>6.7</td>
</tr>
</tbody>
</table>

Table 5: Speedups on 8 nodes

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.

6.4 Performance Analysis

Figure 3 shows the normalized execution time breakdown for the applications we studied. Each bar presents a percentage split-up 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.

For the purpose of this analysis, 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 multi-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 multi-writer support of HLRC helps these applications in achieving good speedups. Water-Nsquared and Radix are multi-writer applications 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.

7 Copy Avoidance

In what follows we investigate the tradeoffs involved in using zero-copy protocols in VIA for software DSM.

Page transfers When the entire shared address space can be registered, zero-copy communication can be achieved for page transfers from home to the non-home nodes. However, any VIA implementation imposes limitations on the amount of memory that can be registered. In order for our DSM system to support problem sizes with large shared memory footprints, we cannot register the entire shared memory region with VIA. Instead, we dedicate a small amount of registered memory just for the buffers used in communication. When the DSM protocol needs to transfer shared data, it copies the data from its actual source location in memory to the communication buffers. Since the shared memory region is not registered with VIA, a similar copy
is required on the receiving side from the communication buffer to the destination address on the receiving node.

To evaluate the impact of the extra copies, we ran all the seven applications with copies at both the send and receive sides. We observed a slight degradation in the performance in terms of an inflated execution time. The page fetch time increases as a result of the additional copies at the home node and the receiving node. The performance degradation was maximum for FFT (15%) and very little (less than 5%) for the other applications.

**Direct Diff**s Diff can be sent either of the following two ways:

- as a request message, requiring the communication thread to receive and process it. In this case, the home node has to explicitly apply the modifications to the shared page. To apply a diff, the home node either needs to unprotect the page for write (in case it is write-protected by the application thread) or it has to use a second mapping to the shared data, dedicated to diff application.

- as an RDMA write directly into the home’s copy of the page. This works only if the entire shared memory region is registered. A send message is generated for every contiguous dirty region of the modified page. Depending on how sparse the modified locations are within the page, and how expensive sending a message is, this approach can suffer potential performance penalties resulting from sending multiple diff messages for a modified page instead of a single one. If a modified page consists of a large number of dirty segments, the overhead of sending a large number of messages will outweigh the benefits of avoiding the copy for the diff.

We looked at two of the applications, viz., Radix and Barnes which generate a substantial amount of diff traffic. 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 modified page. For Barnes, the overhead of sending multiple dirty segments per page outweighs the improvement achieved by avoiding the copy.

8 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 soft-
ware DSM.

A lot of work has been done on shared virtual memory since it was first proposed[22]. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence[13]. RC was used to reduce false sharing by allowing multiple writers [5]. Lazy Release Consistency (LRC) [20, 7] further relaxed the RC protocol to reduce protocol overhead. Treadmarks [19] was the first SVM implementation using the LRC protocol on a network of stock computers. The Automatic Update Release Consistency (AURC) [15] protocol was the first proposal to take advantage of memory-mapped communication to implement an LRC protocol. Home-based Lazy Release Consistency (HLRC) [18] proposed a home-based approach to improve the performance on large-scale machines. Cashmere [21] 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 [14].

The VI architecture [6] builds on previous work in user-level communication. The VI architecture is based on ideas similar to that of U-Net [12], virtual interfaces to the network from application device channels [8], and Virtual Memory Mapped Communication (VMMC) [9]. Other research that discuss user-level direct access to the network interface are FM [25], AM [11], Hamlyn [31], PM [30], and Trapeze Trapeze.

Prototype implementations of the VI Architecture have been developed on Myrinet, and 100 Mb/s Ethernet [1]. M-VIA [23] is a a software emulation of VIA over various network interface cards including stock Ethernet cards. [4] is another implementation of VIA over Myrinet. A performance study of VIA [27] has compared the performance of 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 [3, 29, 2] has looked at exploiting support available in hardware to improve the performance of software DSM. [3] explores performance gains to be obtained from performing asynchronous message handling in the network interface. Another study [29] 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 [2].

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.

9 Software

A software distribution package with the software DSM protocol described in this paper is available for download from our website (http://discolab.rutgers.edu/projects/dsm).

10 Conclusions

We have implemented a high-performance software distributed shared memory protocols for clusters of PC 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. The experimental results show that VIA can be successfully used to support shared memory on cluster of PC but further study is necessary to evaluate its scalability on larger clusters and for a larger set of applications.

References


