

# Multithreaded Home-based Lazy Release Consistency over VIA

Vadim Iosevich and Assaf Schuster  
Technion–Israel Institute of Technology  
Computer Science Dept.  
Technion City, Haifa 32000, Israel  
{vadim\_ds, assaf}@cs.technion.ac.il

## Abstract

*A Distributed Shared Memory (DSM) system is a software or hardware mechanism that provides a distributed application with a shared virtual address space. The efficiency of a DSM system relies mainly on a memory coherency protocol and an efficient communication layer. This article proposes a design for implementing the communication layer on top of the Virtual Interface Architecture (VIA), an industry standard for user-level networking protocols on high-speed clusters. User-level communication protocols operate in a user mode, thus removing the operating system kernel's overhead from the critical communication pass and significantly diminishing communication overhead as a result. We analyze VIA's facilities and limitations in order to ascertain which implementation trade-offs can be best applied to our development of an efficient communication substrate optimized for DSM requirements. We then implement a multithreaded version of the home-based lazy release consistency (HLRC) protocol on top of this efficient substrate. To the best of our knowledge, this is the first multithreaded implementation of HLRC that uses preemptive threads. We evaluate and analyze the performance of this protocol over a wide set of benchmark applications.*

## 1. Introduction

### 1.1. DSM Basics

A *Distributed Shared Memory* (DSM) system is a software or hardware mechanism that provides the distributed application with an abstraction of the shared address space in such a way that all data stored in this space is shared between all nodes in the cluster. Generally, each node uses its local virtual memory as a cache of the shared memory, often identifying the presence of data in the local cache by utilizing the virtual memory hardware. If the data is located on a remote node, the DSM system is responsible for fetching it, while maintaining the correctness of the shared memory. The messages passed by a DSM system are not visible to the application, and this simplifies programming in many ways. Data sharing is implemented through the address space only, and the programmer should not concern himself with the communication protocol of the application. This concept was first proposed by K. Li and P. Hudak in the mid-to-late '80s and implemented in the first software DSM system, named IVY [24, 25].

Actually, the shared data is scattered across nodes in a cluster. In order to keep the cache in a coherent state, the shared data is grained to segments, like lines in a real cache. We call

these portions *coherency units*. A DSM system manages local views in terms of coherency units, which means that either the entire coherency unit is available locally or the entire coherency unit is absent. The size of the coherency unit is called the *granularity level*. The granularity level is one of the main characteristics of a DSM protocol and has a great impact on DSM runtime performance.

A set of rules that defines the state of each coherency unit at every point of time forms a memory coherence protocol. This protocol determines the behavior of the shared memory, and a parallel application running on top of DSM must be aware of this behavior. A formal specification of how memory operations appear to execute to the programmer is called a *memory consistency model*. The consistency model defines restrictions on the values that can be read by an application from the shared memory. It can be said that the consistency model defines, for each memory access operation, when it must be visible to other processors.

## 1.2. Lazy Release Consistency

A consistency model should be effective, and it should provide an intuitive programming paradigm. Effectiveness is generally achieved by alleviating false sharing and reducing communication. Since the introduction of Lamport's now-canonical sequential consistency model [23], various consistency models have been proposed by researchers [13, 21, 19, 5, 6, 18]. The idea of these models is to postpone the propagation of coherence information until synchronization points are reached. This means that between these synchronization points the shared memory may appear inconsistent to different processors. There are two types of synchronization operations, ACQUIRE and RELEASE, used respectively to obtain and yield exclusive access to shared data. These operations can be thought of as standard lock operations.

*Lazy Release Consistency* (LRC) [21, 19] is a refinement of the *Release Consistency* (RC)

model [13]. The RC model requires that shared memory accesses be performed globally upon a RELEASE operation only. The idea of LRC is to make those accesses visible only to the processor that acquires a lock rather than perform all operations globally. False-sharing is alleviated by allowing different processes to access the same page simultaneously if these operations are not synchronized. A home-based implementation of LRC (HLRC) was proposed by Iftode [17]. In this implementation each shared page has an assigned node. This home node always hosts the most updated contents of the page, which can then be fetched by a non-home node that needs an updated version.

## 1.3. Contributions

This article proposes a design for implementation of a communication layer on top of the *Virtual Interface Architecture* (VIA) [36], an industry standard for user-level networking protocols on high-speed clusters. User-level communication protocols operate in a user mode, thus removing the operating system kernel's overhead from the critical communication pass, and significantly diminishing communication overhead as a result. We analyze VIA's facilities and limitations in order to ascertain which implementation trade-offs can be best applied to our development of an efficient communication substrate optimized for DSM requirements.

Our second contribution is an efficient multithreaded implementation of the *home-based lazy release consistency* (HLRC) coherence protocol. To the best of our knowledge, this is the first multithreaded implementation of HLRC that uses preemptive threads. Previous studies proposed non-preemptive multithreading [28] or creating a process for each CPU in an SMP node [32, 7]. Other multithreaded DSM systems do not implement the HLRC memory coherence protocol [35, 31, 26, 27]. All these implementations are described in depth in Section 7.3.

## 2. Virtual Interface Architecture

New high-performance clusters provide a gigabits-per-second bandwidth and extremely low packet transmission latencies [14, 16, 10, 15]. Standard TCP/IP protocols have become slower than the associated hardware, which led to the concept of *user-level networking* (ULN). In ULN, all network operations are performed completely in user mode, without involving the kernel. The *Virtual Interface Architecture* (VIA) [36] was introduced as a standard architecture for user-level networking. One of the main goals of the VIA design was to find a fair trade-off between efficiency and portability, in order to serve a wide range of applications with different types of communication traffic. Consequently, VIA defines a set of low-level network primitives, which greatly complicates the programming if used directly by an application. A medium-level software layer should be implemented on top of VIA to provide the application with high-level network services like sockets, MPI, remote procedure calls, etc. This layer can already be optimized for a specific type of application, taking into consideration communication patterns and application-dependent limitations [11, 2, 4].

VIA provides a communicating process with a directly accessible interface to a *Network Interface controller* (NIC). This is called a *Virtual Interface* (VI) and represents the endpoint of a point-to-point communication channel. The communicating process gets network services by posting work requests to the *Send Queue* and *Receive Queue* of the VI in the form of descriptors. The descriptor is a data structure that holds the full description of the work request, including the transmitted data address and length. Each request is served asynchronously by a NIC in FIFO order. When the status of a descriptor is marked as completed, the descriptor can be safely dequeued from the VI.

VIA exports two basic communication semantics. The first is a Send/Receive model, akin to

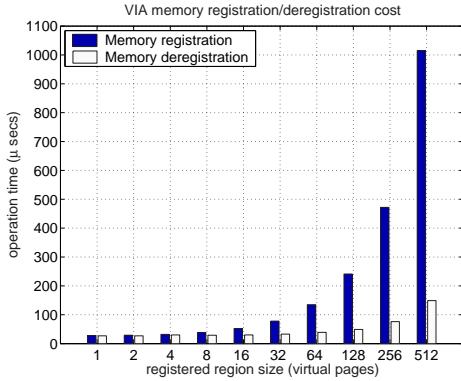
traditional message passing, where both sender and receiver specify the location of transferred data. The sender allocates a buffer, fills it with data to be sent, and posts a descriptor to the Send Queue. The descriptor specifies the buffer's address and length. The receiver allocates an empty buffer where the incoming data will be placed and posts a receive descriptor, containing a pointer to this buffer, to the Receive Queue.

The second communication semantics is the *Remote Direct Memory Access* (RDMA) model. The process that initiates a data transfer operation specifies the source and the destination buffer at the local and remote nodes. There are two types of RDMA operations: RDMA write and RDMA read. An RDMA operation request is posted by the initiator to the send queue just like a regular send request. The difference is that an RDMA operation does not consume a descriptor at the receiver's side.

In both the Send/Receive and RDMA models, the user's data is accessed directly by the NIC. To enable DMA input-output by the NIC, the memory regions participating in the data transfer (descriptors and data buffers) must be registered with VIA. This is in order to maintain a correspondence between virtual and physical addresses. Registered (pinned) pages remain resident in the physical memory until deregistration. Memory registration and deregistration are expensive operations performed by a kernel driver; therefore, the number of these operations should be minimized. The overhead of these operations is given in Fig. 1.

## 3. VIAComm

VIACOMM is a communication protocol that has been designed on top of VIA especially for DSM communication needs. In the following subsections we address the requirements and considerations that prescribe the organization of the VIACOMM library.



**Figure 1. VIA memory registration/deregistration cost**

### 3.1. DSM Traffic Characteristics

Typical DSM network traffic may be divided into three categories as follows:

**Control Messages:** Control messages are designated to request some service or inform another node that some event has taken place. Information carried by a control message usually does not exceed a few of bytes. These messages are generally handled and then immediately discarded. The Send/Receive VIA semantics is most suitable for this type of message.

**State Data:** The HLRC protocol manages complex data structures that must be transferred between nodes. These structures can be transferred through RDMA to avoid buffer copying, but VIA requires that the destination of an RDMA write and the source of an RDMA read be a single virtually contiguous region. Thus, if the data is split into  $n$  segments, then  $n$  RDMA operations must be issued to transfer it. In this case, however, the data may also be packed into a single message and then sent as a regular message in Send/Receive semantics.

**Shared Memory Contents:** The coherency unit

of the HLRC protocol is a memory page. The location of the transferred data at the addressee’s address space is always constant and known to other nodes. Therefore, the data may be transmitted by the RDMA operation, thus avoiding redundant buffer copying.

### 3.2. VIAComm Services

To simplify the design of the communication layer, we leave the decision about the communication semantics to the DSM itself and provide a communicating process with the means to send a regular message of a limited size or to issue an RDMA write operation. If the DSM protocol requires the sending of a regular message whose size exceeds the maximum, it must send a chain of messages and assemble them on the receiver’s side. Each regular message is assigned a type, and the appropriate handler function is asynchronously called by VIACOMM for every regular message. To send a message, the DSM protocol requests a buffer residing in the registered memory and constructs a message in this buffer. The memory participating in the RDMA transfer must be explicitly reserved by the DSM.

### 3.3. Descriptors and Buffer Management

VIACOMM uses two types of descriptors: descriptors for regular messages and RDMA descriptors. All descriptors are preallocated and stored in special pools when not in use. Each node manages two pools, one for regular and one for RDMA descriptors.

The maximal number of RDMA descriptors that may be used simultaneously is limited by the Send Queue length, which defines the size of the RDMA descriptors’ pool. The regular descriptors’ pool supplies descriptors for both the Send and Receive Queues. Each regular descriptor is allocated altogether with a buffer.

When VIACOMM is initialized, it fills the Receive Queues of all VIs with descriptors taken

from the pool. The moment a descriptor has been dequeued from the Receive Queue for processing, a new descriptor from the pool is enqueued instead. VIACOMM tries to keep the Receive Queue as full as possible to enable continual communication. When an incoming descriptor is processed by a handler function, it is returned to the pool if not reused by a handler.

### 3.4. Flow Control

Two conditions must be satisfied to post a regular descriptor to a Send Queue: (i) the Send Queue must not be full and (ii) the corresponding Receive Queue must have a descriptor with a buffer large enough to receive this message. To safely post an RDMA descriptor, only the first condition must be satisfied because no descriptor will be consumed in the Receive Queue. VIACOMM implements a credit-based flow control protocol that grants a number of “credits” to the VI, and each credit allows one descriptor to be sent: regular or RDMA. When there are no more credits, no more descriptors can be posted to the Send Queue.

Credits spent for RDMA descriptors are returned to the VI when the RDMA descriptor is completed and dequeued. A credit spent for sending a regular message may be returned only when a new descriptor has been posted to the Receive Queue in place of the one consumed by this message. That means that this type of credit can only be returned by an addressee node. Returned credits are piggybacked by descriptors flowing in the opposite direction; if there is no traffic that can piggyback returned credits, an empty message is explicitly sent when a critical number of credits has been accumulated by the receiver.

### 3.5. Handling Asynchronous Messages

When VIACOMM is initialized it creates a server thread, which waits in an infinite loop for completed descriptors. The sooner a completed descriptor is dequeued from the Receive Queue, the sooner a new descriptor is posted in

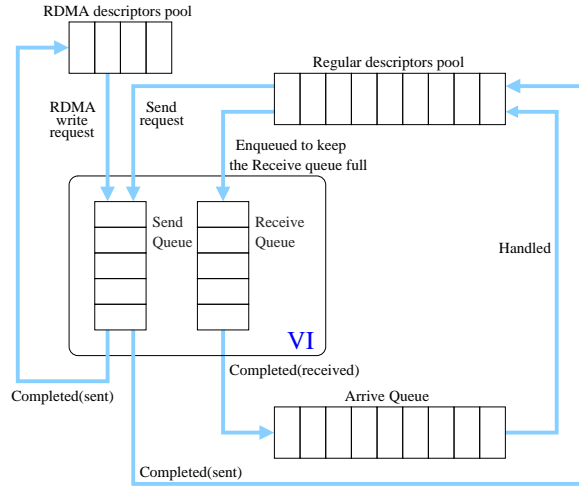


Figure 2. VIACOMM descriptor management

its place. This, in turn, enables the returning of a credit to a sender’s VI. Therefore, extracting completed descriptors is a high priority task. All completed descriptors are transferred to the “Arrive Queue,” where they wait to be handled. This queue is processed when there are no more completed descriptors to dequeue from the VI. The entire mechanism is illustrated in Fig. 2.

The server thread consists of two Win32 fibers. The poller fiber extracts completed descriptors from VIs and the dispatcher fiber handles the Arrive Queue in FIFO order. Message handling may involve send operations; therefore, the handler may be blocked by the flow control protocol. In this case the control is switched back to the poller fiber. Implementing the poller and the dispatcher as two fibers within one thread saves synchronization and context switch overhead.

### 3.6. Performance Evaluation

We have conducted a number of tests to measure VIACOMM performance. All tests were performed on two computers from our cluster. The testbed environment is outlined in Section 5.1. We tested four operational modes:

**Send/Receive without data copying:** There is no message copying on the receiving node.

All messages are discarded immediately upon receiving.

**Send/Receive with data copying:** All received data is copied to a temporary buffer and then discarded. The overhead of data duplication increases the latency and consequently decreases the throughput.

**RDMA write with immediate data:** The RDMA descriptor contains a boolean flag, which, if set, forces the RDMA operation to consume a descriptor in the Receive Queue at the targeted node. This descriptor is marked as complete when the RDMA operation finishes. This mode enables the addressee node to recognize that the RDMA operation has been completed. Although VIACOMM does not currently use this mode, we test it in order to compare it with the RDMA mode that we used in our system.

**RDMA write without immediate data:** No descriptor is consumed in the Receive Queue by an RDMA operation. Consequently, the targeted node cannot synchronize on this RDMA operation.

The resulting one-way latency and throughput graphs are shown in Fig. 3 and Fig. 4 respectively. The latency increases linearly in accordance with message size for all operational modes, and the best latency is achieved for RDMA write operations without immediate data. For small messages, the overhead of processing a descriptor seriously affects the throughput. The larger the transmitted message, the better the throughput.

#### 4. A Multithreaded HLRC Protocol

This section describes a multithreaded Multiple-Writers/Multiple-Readers memory coherence protocol that guarantees Lazy Release Consistency [19] semantics for distributed shared memory. The protocol is based on Home-based

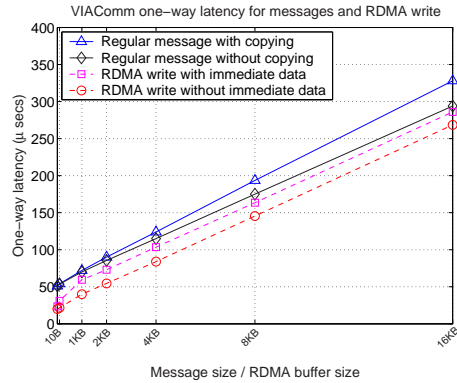


Figure 3. VIAComm one-way latency

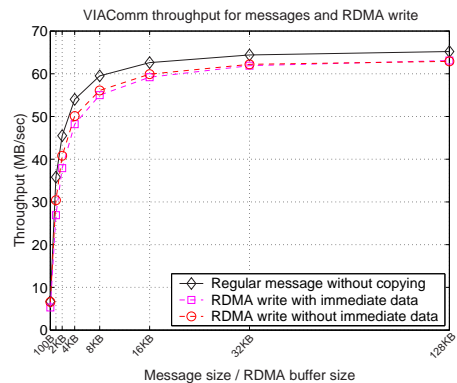


Figure 4. VIAComm throughput

Lazy Release Consistency [17, 38] and its extension for SMP clusters [7, 8]. To preserve LRC memory semantics, a thread that performs an ACQUIRE operation invalidates all pages which were modified on remote nodes and those modifications must become visible to this thread before the ACQUIRE. A subsequent reference to an invalidated page causes a page fault, and the missing page is fetched from the home node, which always has the most updated copy of the page. After the page contents have been updated, the interrupted shared memory access can safely proceed.

#### 4.1. Two-level Memory Coherence Protocol

The need to support multithreading complicates the memory coherence protocol. From the application's point of view, the shared memory must remain coherent between all threads regardless of their physical location in the cluster. Our coherence protocol has been designed to take advantage of the address space sharing between Win32 threads provided by the operating system. Hence, the shared memory coherence will be managed at two levels: an intra-node level between threads on the same node and inter-node level between different nodes in the cluster. Henceforth, we will denote threads running on the same node as *local threads*, whereas threads running on different nodes will be denoted as *remote threads*.

Ideally, invalidations performed by a thread should not be visible to other local threads running on this node. To implement this functionality, each thread should have its own view of the shared memory. Unfortunately, the views cannot be separated in the implemented model because all threads reference shared data through the same virtual addresses, and the virtual address space is shared between local threads. Thus, if one thread invalidates a virtual page, all other local threads see this invalidation even though the consistency model does not require that they do so. This does

not violate the LRC semantics but may lead to redundant protocol actions and messages.

#### 4.2. Granulating the Execution Time

In the protocols based on the LRC semantics the execution time is segmented into *intervals*—time segments delimited by synchronization events. This approach allows the grouping of memory accesses that must be performed with respect to other nodes. All shared memory modifications performed during the interval must be recorded in order to be reported later to nodes synchronizing with this one. To track write accesses, all pages must be write-protected when an interval starts. The first write access switches the page to a *dirty* state, removing the write-protection. At the same time, the DSM system saves a copy of the page, called a *twin*. When an interval is finished, all modified pages are differentiated from their twins, and structures called *diffs*, containing all modified words, are flushed to corresponding home nodes.

This operation is relatively expensive and, if performed frequently, may degrade the performance. Because local threads use the same shared memory view, it is impossible to track local threads' write operations independently. If one thread switches the page to the dirty state, all subsequent writes to this page, issued by other local threads, will be transparent for the DSM system and therefore untrackable. Therefore, intervals cannot be managed for each thread separately and must be handled on a per-node basis.

To preserve the LRC semantics, an interval on a node  $P$  must be finished before any remote thread can request any modifications that were performed in this interval. A remote thread may require modifications performed in this interval only after it acquires a lock previously released by some  $P$ 's thread. A thread that acquires a lock invalidates pages that were modified by other nodes in previous intervals. However, the invalidated page may be in a dirty state, and thus it cannot

be updated from the home node until all pending modifications are reported to that home. Otherwise all those pending modifications will be lost.

The first solution to this problem is to finish an interval when a thread acquires a lock. Thus all modified pages will be differentiated and no dirty page will be invalidated. This technique was used in the first HLRC implementation [17]. We henceforth refer to this scheme as *aggressive intervals*. This solution increases the number of intervals, thus reducing the protocol overhead.

The second solution is a *2-way-diffing* technique proposed for the Cashmere-2L DSM system [32]. We slightly modify this technique to enable the dirty page to be fetched from the home node, and sent, by means of an RDMA operation, directly to the right location. Hence, a dirty page is updated as follows: (1) It is differentiated from its twin and the resulting diff is locally saved. (2) A new version of the page is fetched from the home node and sent directly to the correct location. (3) The updated version is copied to the twin. (4) The locally saved diff is applied to the dirty page.

This technique makes it possible to finish an interval only when lock ownership is granted to a remote node, but it obviously adds some overhead to the protocol. When we compared this *lazy intervals* scheme with the aggressive interval scheme, we found that the former is preferable for all the tested applications. The results showing the performance gain from the lazy intervals scheme are presented in Table 1.

### 4.3. Recording Memory Accesses

To record shared memory modifications we use a *bins* data structure similar to the one described by Bilas in [7, 8]. This is a global database, partially replicated at each node. The bins data structure contains stacks for each of the participating nodes. When any local thread issues the first modification to a non-dirty page within an interval, it pushes a record describing this event

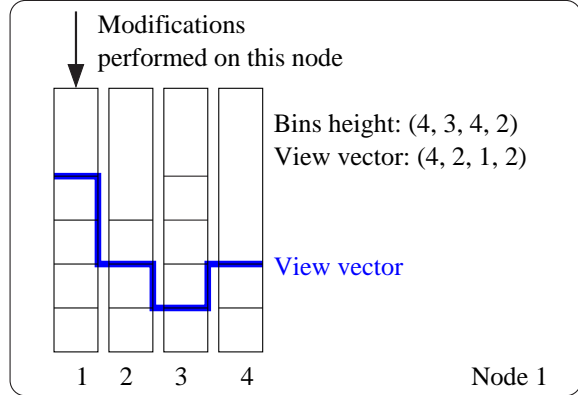


Figure 5. A typical bins structure for four nodes

to the stack corresponding to its node. Bins can be thought of as the partial knowledge that a node has about all other nodes in the system. Obviously, the same entry in the bins always holds the same information on every node. This information flows between nodes on synchronization operations. A node that sends the bins data to a remote node always sends all recently available entries, those that are still not available on that remote node. Even so, not all of these entries are currently needed by the remote node; only those which must be visible before the current ACQUIRE operation are actually required. Therefore, the node applies only a subset of the bins at a particular time. This subset is bounded by a *view vector* where the  $i$ -th element corresponds to the  $i$ -th stack of the bins and stores the height of the stack that has been applied so far. All entries above the view vector are the result of prefetching. These write notices will be applied when the view vector rises (i.e., when the values of its elements increase). Fig. 5 illustrates a typical bins replica in a system consisting of four nodes.

It is clear that the bins database has no a size limit and can easily consume enormously large space. Nevertheless, bins content that has already been applied and is known to be present at all other nodes can safely be discarded because this

Application	Runtime (in seconds) on twelve nodes					
	One thread per node			Two threads per node		
	Aggressive intervals	Lazy intervals	Perf. gain	Aggressive intervals	Lazy intervals	Perf. gain
Water-nsq	168.05	166.2	1.10%	95.97	93.69	2.38%
Water-sp	77.94	77.84	0.13%	54.22	54.08	0.15%
LU	46.59	34.61	25.71%	49.88	40.63	9.64%
FFT	48.22	47.70	1.08%	47.41	47.27	0.15%
TSP	244.59	235.48	3.72%	135.77	131.89	4.04%
SOR	41.44	41.03	0.99%	37.13	36.97	0.17%
Barnes-sp	4.69	4.36	7.04%	4.91	4.34	0.59%
Radix	7.56	7.48	1.06%	5.61	5.53	0.08%
Volrend	21.55	19.81	8.07%	13.48	12.31	1.22%
Ocean	50.81	50.81	0%	49.17	49.17	0%
NBody	16.74	16.68	0.36%	10.45	10.38	0.07%
NBodyW	19.74	19.72	0.10%	13.76	13.74	0.02%

**Table 1. Comparison of two interval finishing policies**

information will never be used again. To cut unused segments of bins, a garbage collector would have to be implemented. To do away with the necessity for garbage collection, we implement bins stacks as cyclic buffers, such that new pushed entries overwrite the old ones. Each entry in the bins has a logical index that grows incrementally and is translated to a real index in the buffer by a modulo operation. This operation can be efficiently implemented by applying a bitwise mask if the buffer’s size is a power of two. If some node tries to use more than the allocated stack size, we will know that useful data is being overwritten. A notable disadvantage of this approach is that if the size of the preallocated cyclic buffers is insufficient, runtime failure may result. We discovered that for all of the benchmarks used in our experiments, we need to allocate 1664KB for each stack. Thus, the entire bins database occupies 19.5MB for 12 nodes. This approach is inappropriate for larger clusters, and for applications that consume more time and memory than the applications we tested. In such cases, more complex data structures and more complex algorithms must be implemented.

#### 4.4. Data Transfer Trade-offs

Diffs and bins may be transferred by RDMA operations or as regular messages. VIA requires that the RDMA write destination be a single contiguous buffer; that is, if the diff is flushed through the RDMA, a separate RDMA operation should be issued for each contiguous sequence of modified blocks. In contrast, packing diffs into regular messages incurs an overhead over and above that of immediate copying. The packing requires that an index be attached to each contiguous sequence of blocks, identifying its location in a page. It should be noted that all diffs flushed to the same home node require that at least one regular message be sent, informing the home node which pages were updated. A packed diff may be piggybacked by this message. If the packed diff exceeds the maximal message size that can be carried by a single descriptor, it should be split into several messages and sent in a chain. For the 12 tested benchmarks, only FFT showed better performance for RDMA diffing. The others perform better when diffs are packed and sent as regular messages.

Bins can also be sent through RDMA or can be packed in chains of regular messages. When a portion of the bins is sent to a remote node, it is always attached to some control message sent by the coherence protocol. This message can piggy-back the bins contents. This approach is preferable when the contents of the stacks of several bins is not large and can be packed into a single descriptor. If the contents are sent through RDMA, a separate operation must be issued for every stack, because only within a single stack can the data be organized continuously. Nevertheless, even in the worst case, the number of RDMA operations will be the number of bins stacks minus one. We found that 9 out of the 12 tested benchmarks perform better when the contents of the bins are sent through RDMA write operations. The results are summarized in Table 2.

#### 4.4.1 Fetching the Updated Shared Data

We use vector timestamps to indicate the version of a page that must be obtained by a node in order to proceed with the execution. This approach is detailed by Iftode in [17]. Multithreading support requires a page fetching protocol that is more complex than in the single-threaded version. While a thread waits for an update from the home node, a more recent write notice for this page can arrive as the result of an ACQUIRE operation performed by another local thread. If this happens, then the page request must be resent in order to inform the home node that a more recent version is required. If the home node receives this second request after it has served the first one, it will send the page again, and the requesting node will receive two copies. Therefore, each copy of the page must be followed by a version timestamp. This timestamp would then allow the node to discard unneeded copies.

## 5. Performance Evaluation

### 5.1. Experimental Platform

Our testbed environment is a cluster of twelve SMP machines with two 733MHz P-III processors and 512MB physical memory. All nodes run the Windows 2000 Workstation operating system. The cluster is interconnected by the ServerNet-II [16] high-speed VIA-oriented network. The DSM uses the VIACOMM communication substrate, described in Section 3.

### 5.2. Benchmark Application Suite

Our benchmark application suite consists of two microbenchmarks, NBodyW and NBody, eight applications from the SPLASH-2 [37] benchmarks suite (Barnes, Volrend, LU, Watersnq, Water-sp, FFT, Radix and Ocean), as well as TSP and SOR from the TreadMarks [22] benchmark applications.

NBodyW is a microbenchmark that imitates a kernel of n-body applications. The program operates with a large set of 64-byte bodies and performs three phases as follows: (1) Each of the  $P$  application's threads reads the entire set of bodies. (2) Each of the  $P$  application's threads processes and updates  $1/P$  of the bodies. The processing of a body is simulated by a constant-length busy loop. (3) A single thread updates all the bodies (sequential phase). NBody is a shortened modification of NBodyW that contains only the first two phases. Hence, this application contains one coarse phase and one fine phase.

A detailed description of the other benchmarks can be found in related papers and is not provided here for lack of space. Table 3 summarizes input data sets and memory sharing characteristics of all tested benchmarks.

The speedups achieved by all the applications in both single-threaded and multithreaded modes are summarized in Fig. 6. The execution time breakdown for a single-threaded mode is shown in Fig. 7.

Application	Optimal bins transfer strategy	Optimal diff transfer strategy
Water-nsq	RDMA	message
Water-sp	RDMA	message
LU	message	message
FFT	RDMA	RDMA
TSP	RDMA	message
SOR	message	message
Barnes-sp	RDMA	message
Radix	RDMA	message
Volrend	message	message
Ocean	RDMA	message
NBody	RDMA	message
NBodyW	RDMA	message

**Table 2. Optimal strategies for bins/diff transfer on twelve nodes**

Application	Input data set	Shared memory	Sharing granularity	Synch
Water-nsq	8000 molecules	5.35MB	a molecule (672B)	B, L
Water-sp	8000 molecules	10.15MB	a molecule (680B)	B, L
LU	$3072 \times 3072$	72.10MB	block (coarse)	B
FFT	$2^{20}$ numbers	48.25MB	a row segment	B
TSP	A graph of 32 cities	27.86MB	a tour (276B)	L
SOR	$2066 \times 10240$	80.73MB	a row (coarse)	B
Barnes	32768 bodies	41.21MB	body fields (4–32B)	B, L
Radix	10240000 keys	82.73MB	an integer (4B)	B, L
Volrend	a file “head.den”	29.34MB	a $4 \times 4$ box (4B)	B, L
Ocean	a $514 \times 514$ grid	94.75MB	grid point (8B)	B, L
NBody	32768 bodies	2.00MB	a body (64B)	B
NBodyW	32768 bodies	2.00MB	a body (64B)	B

**Table 3. Benchmark characteristics. B stands for barriers, L stands for locks**

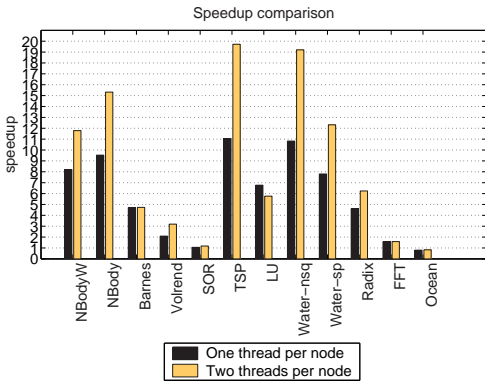


Figure 6. Multithreaded HLRC speedup, measured as the relation between the serial execution time and the most effective execution time on a 12-node cluster.

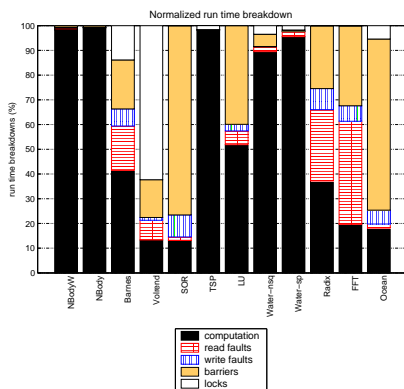


Figure 7. Multithreaded HLRC runtime breakdown, measured on 12 nodes when each node runs one application thread.

**NbodyW** is not fully parallelized because the execution time of the first and third phases is independent of the number of threads being executed. Nevertheless, the absence of both synchronization and false-sharing within phases results in an extremely low protocol overhead and a speedup of 8.21. If several threads run on a node, only the second phase can benefit from this additional parallelism. Multithreading decreases the number of write faults because local threads predict write faults for one another. Moreover, additional computation power causes the second phase to finish faster. The second thread improves the performance by 43.5%.

**Nbody** achieves a speedup of 9.53, which is 16% better than that of NBodyW. This speedup is due to the abolishment of the sequential phase, which also allows NBody to benefit from multithreading to a larger extent than NBodyW and improve its performance by 60.8

**Barnes** is characterized by an irregular fine-grain access to bodies and the resulting speedup is 4.71. Running two threads per node does not improve the performance. The number of read faults remains unchanged because threads simultaneously try to read the same data and therefore do not prefetch data for one another. The number of write faults also remains unchanged. Large protocol overhead and increased memory contention do not allow Barnes to benefit from the additional computation power provided by multithreading.

**Volrend** implements distributed task queues and performs an irregular fine-grain access as part of a task. Synchronizing on a task queue causes significant lock overhead. Frequent synchronization causes a large increase in the number of coherency operations. As a result, Volrend does not perform well. Another factor that limits the performance is the sequential phase of writing an output image file. These factors result in a speedup of 2.09% only, and the additional thread improves the performance by 52% for a total speedup of 3.19.

**SOR** performs a number of phases separated by barriers. The access pattern in each phase is coarse-grain and sequential—each application’s thread reads and writes a large contiguous set of data, but only the data placed on the boundaries between different threads is actually shared. According to the consistency model requirements, all threads synchronize all their modifications on every barrier even though they are not actually required to do so for this specific application. This causes an enormous number of unnecessary coherency operations. Additional overhead is also caused by the fact that a thread finishes an interval after each phase, which results in the *Read-Only* permission being set for all recently updated pages. This in turn causes a write fault for each page at the next update. Hence, the resulting speedup is only 1.06. Adding a thread on each node hardly improves the application’s speedup. Now each node reads and writes two contiguous blocks of rows. These blocks are not adjacent because an application cannot consider the threads’ placement in a cluster. Consequently, the total length of shared boundaries is doubled. The number of write faults does not change because a write fault is generated for all accessed pages and not only for boundaries, while the number of read faults is doubled.

**TSP** exhibits an extremely high computation-communication ratio and thus achieves a speedup of 11.05. Low protocol overhead makes the application scalable for multithreading, since all of the computation power of the additional second thread is used entirely for the computation. This additional thread results in an almost double speedup of 19.72.

**LU** is characterized by the coarse-grain access and significant barrier overhead that are caused by the sequential phases. Most of the coherency operations performed on a barrier are wasteful because threads do not need to see the modifications performed by all other threads. LU does not benefit from multithreading at all. One reason is that the number of page faults stays the same (be-

cause each node still updates the same portion of the data). In addition, there is no communication-computation overlapping because both threads simultaneously wait on a barrier and both start intensive calculations at the same time. Moreover, a node that must serve a data request is itself busy with computations, and the additional thread overloads the CPU, reducing the availability of the server thread. This increases the average time of waiting on a read fault from 266 to 6454  $\mu\text{secs}$ . The average time of waiting on a write fault increases from 59 to 72  $\mu\text{secs}$ . As a result, the speedup worsens by 14.9%, decreasing from 6.77 to 5.76.

**Water-nsq** performs irregular fine-grain access to molecules, but infrequent synchronization results in a low protocol overhead. This allows Water-nsq to achieve a good speedup of 10.82 and also makes it scalable for multithreading. Its scalability is due to the fact that the majority of the execution time is spent on computation and thus can benefit from additional CPU. Multithreading improves HLRC’s speedup by 77.5%.

**Water-sp** aggregates molecules in space cells (boxes). Water-sp achieves lower speedups than Water-nsq, because the work distribution in Water-sp is not perfect and limits the maximal speedup. For our data set, which consisted of 8000 molecules, a processed space is represented as a  $5 \times 5 \times 5$  matrix of boxes. These boxes must be distributed among threads and the distribution cannot be completely fair. The HLRC protocol reaches a speedup of 7.80 and 12.32 respectively, in single-threaded and multithreaded modes.

**Radix** combines both coarse-grain and fine-grain access patterns. The histogram building and key permutation phases perform efficiently because there is no synchronization within phases. Nonetheless, the collecting of the histograms and the existence of inter-phase barriers add overhead to the protocol. Because radix does not perform enhanced computations, the overhead for the protocol is relatively large and a speedup of only 4.61 is attained. Adding a second computation thread

for each node improves the HLRC protocol performance by 35.4%, with a speedup of 6.24.

**FFT** is characterized by a coarse pattern of memory accesses. A low computation-communication ratio does not allow the application to achieve a good speedup. During the matrix transposition phase, each thread reads a row's segments from other threads. The HLRC protocol has a significant barrier overhead, since all threads exchange information about all their modifications on a barrier after the transposition phase. FFT does not benefit from multithreading. Not only does multithreading not decrease the number of write faults; it also doubles the number of read faults because two local threads read two disjoint segments of columns. Computation that can benefit from the additional computation power comprises only a small part of the execution time.

**Ocean** is characterized by a large barrier overhead that prevents it from achieving good speedups. For sequential executions of less than a minute, it passes more than two thousand barriers. This enormous number of barriers results in an overhead of 69.2% and does not allow the application to achieve any speedup. Thus multithreading offers no benefit in this case.

## 6. Conclusions

We have developed a portable and efficient communication layer on top of VIA, which is optimized for the DSM traffic pattern. This middleware makes use of a zero-copy data transfer protocol to minimize communication overhead. Nevertheless, RDMA semantics poses restrictions on the placement of the transferred data: fragmented data cannot be transferred efficiently. We have found that there is no single communication semantics that is optimal for transferring the coherency state information for all applications. The less fragmented the data to be transferred, the more efficiently the RDMA semantics will operate.

We have presented a multithreaded variation of the HLRC memory coherence protocol. We outline the protocol structure and major trade-offs and considerations in its design. The protocol was tested on a set of twelve benchmarks and the average speedup of the single-threaded mode is 5.97. We show that, while not always beneficial, multithreading improves the average runtime performance of the protocol by 35.5%.

## 7. Related Works

### 7.1. Other DSMs Implemented over VIA

To the best of our knowledge, there are two other DSM systems that use VIA as a communication layer. Rangarajan and Iftode [29] present a DSM that supports the HLRC memory model and runs on a GigaNet VIA-based network. Each node executes only one application thread. Pages and diffs are transferred between nodes using RDMA write operations. Requests that require a response are sent using the Send/Receive semantics while messages that do not require a response are sent using the RDMA semantics. This limits the number of descriptors that must be preallocated. This design is not suitable for our system because we choose the most efficient semantics for transferring bins and diffs and do not limit the DSM to just one of the available semantics. Messages that do not require a response and are sent through the RDMA write are consumed in a busy loop by the application thread, since there is nothing else the CPU can do. Obviously, polling for a message in a busy loop is not feasible in a multithreaded DSM.

The second implementation is a porting of TreadMarks [22, 1] to VIA [3]. The goal of this work was to design a new communication substrate without performing any major modifications to the TreadMarks layer, which originally uses UDP-based communication. Consequently, the communication services required by TreadMarks are well-suited to UDP/IP primitives. This

means the protocol can send messages from and receive them at any buffer, which implies that data can be copied between VIA's registered memory and TreadMarks' buffers. Furthermore, RDMA semantics is not used to transfer large portions of data. In contrast to our DSM, TreadMarks relies on a request-reply type of communication. This leads to the concept of using two different VIs for establishing request and response communication channels between each pair of nodes. This also limits the number of descriptors that can be used simultaneously and eliminates the need for complex flow-control.

## 7.2. User-level Communication Substrates

Other than VIA user-level communication substrates were used to implement DSMs. Bilas [9] used a VMMC [12] communication layer to examine how the DSM performance can be improved by protocol and communication co-design. He used a NIC support to decouple asynchronous message handling from protocol processing and eliminate the need for expensive interrupts or polling in the DSM protocol. The asynchronous message handling is done entirely in the NIC, and the host CPU performs protocol processing at synchronous points only. The HLRC protocol, used in the study, uses a mechanism similar to RDMA to transfer both diffs and bins.

Stets et al. [33] use the Memory Channel [15] communication substrate to implement the Cashmere DSM system. Memory Channel provides remote-write capability, which allows to modify remote memory without remote CPU intervention. It also supports a total event ordering and broadcast that was proven to be effective techniques to improve the DSM performance.

Talmor [34] introduces the new approach of integrating a DSM system with a fast communication layer by offloading DSM management activities from the node CPU to the NIC CPU. The smart NIC has a direct memory access to the ad-

dress space of the communicating process and handles messages at a network layer saving the polling or interrupt cost.

## 7.3. Multithreaded DSMs

Rangarajan et al. [28] introduced a multithreaded HLRC protocol that uses non-preemptive threads. This technique does away with internal protocol synchronization overhead and simplifies the design. Nevertheless, it hardly utilizes the computational power provided by SMP machines. The reported performance improvement is 15–30% in three out of four applications studied.

Thitikamol and Keleher [35] presented the results of adding non-preemptive multithreading to the CVM [20] DSM system, which implements the LRC memory semantics. Non-preemptive multithreading resulted in speedup improvements of at least 20% in two of the tested applications, and better than 15% for several other applications.

Speight and Benett [31] studied the use of preemptive multithreading in the Brazos [30] DSM system, which implements a scope consistency protocol. The average performance improvement was reported to be 17%.

Several studies were carried out to investigate the intra-node level of coherency support within clusters of SMPs. Developers of Cashmere-2L [32] propose a two-level “moderately lazy” coherence protocol that takes advantage of data sharing within SMPs.

Bilas [7, 8] expands the HLRC model to run on an SMP cluster and investigates the performance impact of SMP computers on the HLRC model. These models do not use a multithreading facility but rather run a process on each CPU of the SMP machine.

## References

- [1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel.

- Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
- [2] O. Aumage, L. Bougé, and R. Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607–626, Apr. 2002.
- [3] M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan. Implementing TreadMarks over Virtual Interface Architecture on Myrinet and Gigabit ethernet: Challenges, design experience, and performance evaluation. In *International Conference on Parallel Processing (ICPP '01)*, Valencia, Spain, pages 167–174, September 03 - 07 2001.
- [4] A. Begel, P. Buonadonna, D. E. Culler, and D. Gay. An analysis of VI Architecture primitives in support of parallel and distributed communication. *Concurrency and Computation: Practice and Experience*, 14(1):55–76, 2002.
- [5] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.
- [6] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l. Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.
- [7] A. Bilas. *Improving the Performance of Shared Virtual Memory on System Area Networks*. PhD thesis, Dept. of Computer Science, Princeton University, Nov. 1998.
- [8] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. *IMA Volumes in Mathematics and its Applications*, 105, 1998.
- [9] A. Bilas, D. Jiang, and J. P. Singh. Accelerating shared virtual memory via general-purpose network interface support. *ACM Transactions on Computer Systems*, 19(1):1–35, 2001.
- [10] 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*, 15(1):29–36, 1995.
- [11] L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine: An efficient and portable communication interface for RPC-based multithreaded environments. In *IEEE PACT*, pages 240–247, 1998.
- [12] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software support for virtual memory mapped communication. In *Proc. of the 10th Int'l. Parallel Processing Symp. (IPPS'96)*, 1996.
- [13] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.
- [14] Gigaset, Inc. Gigaset cLAN Family of Products. <http://www.gigaset.com>, 1997.
- [15] R. Gillett, M. Collins, and D. Pimm. Overview of Memory Channel network for PCI. In *Proceedings of the 41st Annual IEEE Computer Society Computer Conference, Santa Clara, CA*, pages 244–249, Feb. 1996.
- [16] A. Heirich, D. Garcia, M. Knowles, and R. Horst. ServerNet-II: A reliable interconnect for scalable high performance cluster computing. Technical report, Compaq Computer Corporation, Tandem Division, Sept. 1998.
- [17] L. Iftode. Home-based shared virtual memory (thesis). Technical Report TR-583-98, Princeton University, Computer Science Department, June 1998.
- [18] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, 1996.
- [19] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, Computer Science Department, Jan. 1995.
- [20] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l. Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, 1996.

- [21] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l. Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.
- [22] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.
- [23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
- [24] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing (ICPP'88)*, volume 2, pages 94–101, Aug. 1988.
- [25] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *ACM Transactions on Computer Systems*, pages 17(4):321–359, Nov. 1989.
- [26] M. D. Marino, G. L. de Campos, and L. M. Sato. An evaluation of the speedup of Nautilus DSM system.
- [27] M. C. Ng and W. F. Wong. ORION: An adaptive home-based software distributed shared memory system. In *ICPADS*, pages 187–194, 2000.
- [28] M. Rangarajan, S. Divakaran, T. D. Nguyen, and L. Iftode. Multi-threaded home-based LRC distributed shared memory. In *The 8th Workshop of Scalable Shared Memory Multiprocessors (held in conjunction with ISCA)*, May 1999.
- [29] M. Rangarajan and L. Iftode. Software distributed shared memory over virtual interface architecture: Implementation and performance. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, Georgia, USA*, October 10-14 2000.
- [30] W. E. Speight and J. K. Bennett. Brazos: A third generation DSM system. In *Proc. of the USENIX Windows NT Workshop*, 1997.
- [31] W. E. Speight and J. K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 312–322, 1998.
- [32] R. Stets, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. L. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Symposium on Operating Systems Principles*, pages 170–183, 1997.
- [33] R. Stets, S. Dwarkadas, L. Kontothanassis, and M. L. Scott. The effect of network total order, broadcast, and remote-write capability on network-based shared memory computing. In *Proceedings of Sixth International Symposium on High-Performance Computer Architecture*, pages 265–276, Toulouse, France, January 08–12 2000.
- [34] Y. Talmor. Higher-performance DSM by peaking network utilization. Master's thesis, Computer Science Department, Technion–Israel Institute of Technology, Haifa, Israel, Apr. 2003.
- [35] K. Thitikamol and P. J. Keleher. Per-node multithreading and remote latency. *IEEE Transactions on Computers*, 47(4):414–426, 1998.
- [36] Virtual Interface Architecture Specification. Version 1.0. <http://www.viarch.org>, Dec. 1997.
- [37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.
- [38] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.