Evaluation of RDMA opportunities in an Object-Oriented DSM

Ronald Veldema and Michael Philippsen

University of Erlangen-Nuremberg, Computer Science Department 2,
Martensstr. 3 • 91058 Erlangen • Germany
{veldema, philippsen}@cs.fau.de

Abstract. Remote Direct Memory Access (RDMA) is a technology to update a remote machine's memory without intervention at the receiver side. We evaluate where RDMA can be usefully applied and where it is a loss in Object-Oriented DSM systems. RDMA is difficult to use in modern OO-DSMs due to their support for large address spaces, advanced protocols, and heterogeneity. First, a communication pattern that is based on objects reduces the applicability of bulk RDMA. Second, large address spaces (meaning far larger than that of a single machine) and large numbers of machines require an address space translation scheme to map an object at different addresses on different machines. Finally, RDMA usage is hard since without polling (which would require source code modifications), incoming RDMA messages are hard to notice on time.
Our results show that even with RDMA, update protocols are slower than invalidation protocols. But RDMA can be successfully applied to fetching of objects in an invalidation protocol and improves performance by 20.6%.

1 Introduction

A Software Distributed Shared Memory (S-DSM) system allows for easy distributed programming by making a cluster seem like a single, big computer. The current proliference of Java programmers increases the importance of Java DSMs.

Recent cluster interconnects can directly and efficiently read/write another machine's memory by means of explicitly programmed Remote Direct Memory Access (RDMA). Note that an RDMA operation is performed without cooperation from the receiving machine, except for the initial setup of RDMA-able memory spaces. With Infiniband, RDMA can be up to 6-10 times faster than send/receive based primitives. For example, using 3Ghz CPUs, a 1 byte RDMA costs about 2 μs whereas a normal message send (including protocol processing to deliver the packet to the application layer) takes 17 μs. It therefore seems promising to employ RDMA in DSM systems to implement memory consistency protocols. However, due to a number of restrictions and the lack of message receipt notification, protocols can become more complex, so that performance is reduced.

Java DSMs must implement the Java memory model using some memory consistency protocol. There are two basic memory consistency protocols: invalidation protocols and update protocols. In an invalidation protocol, a machine asks a 'data-owning' machine (called the home-node), to send over the requested data (*fetch*), and caches it locally until it is invalidated (and sent back, i.e., *flushed*). In an update protocol, a machine *broadcasts* its changes to all (accessing) machines so that a 'write' to some data causes communication while a read of some data will not cause communication. To reduce communication load, changes can be aggregated for delayed bulk broadcasts till a later synchronization action. Update protocols can be implemented solely by means of RDMA, i.e., without any explicit messaging.

Below, this paper investigates the RDMA opportunities in an invalidation protocol and in two update protocol alternatives.

Our prototype implementation uses Jackal [9, 10], a Java based S-DSM system, because of its support for plugable DSM protocols and its simple mark-and-sweep Garbage Collector (GC) that makes RDMA-based DSMs easier to implement (as objects do not move during a program's runtime). Jackal compiles a Java program directly to an optimized native executable. We currently have code generators for x86, AMD64, PowerPC, and IA64. Any multi-threaded Java program can as such run *without change* on a cluster of workstations. Where some DSM systems transfer MMU pages over the network (fixed 1 to 4 Kbyte chunks of memory), Jackal's granularity is a region: a single Java object or a 64 KByte chunk of an array. This automatic array chunking reduces the possibility of false sharing.

Finally, Jackal is very flexible. It supports a very large virtual address space where each machine adds its memory to the global pool without limits on the number of participating machines. This large address space is one of the main obstacles for RDMA use. To build this large address space, it is necessary to translate object references between machines. This address translation scheme works as follows. At object allocation, each object is assigned a cluster-wide unique Global Object Reference (GOR). When sending an object reference to another machine, that objects's GOR is sent instead, combined with a type-descriptor structure of the object pointed to. When the target machine receives the GOR, it consults a hash table to see where the object's local copy is allocated. If no local copy has been allocated yet at that machine, the type-descriptor is used to locate the local machine's meta-class instance for that object. That meta-class contains enough information (type, size in memory, etc.) to allocate the local copy and store the new pointer in the table. Afterwards the local copy's address is used.

This scheme therefore has the following characteristics: copies of objects at different machines have different addresses, each reference sent over the network needs to be translated, and finally, copies are allocated lazily so that we cannot be sure that for each object a local copy always exists before sending references to other machines. These features make an RDMA based implementation harder.

2 Related work

Both invalidation and update protocols exist in many variations and can operate under different memory models. For an (older) overview of DSM systems see [8].

To our knowledge, only few attempts have been made to use RDMA in a DSM protocol. In each of [7, 2, 5] a page oriented Home-based Lazy Release Consistency protocol (HLRC) is optimized with RDMA. In [7] multiple page diffs (the changes made by the local processor) are sent to their home via RDMA. In [2] diffs are applied by RDMA to the home-node copy. Our system, however, is not page based but object based. Also, we examine the opportunities offered by RDMA for both invalidation and update protocols. In [5] it is investigated how to allow multiple threads per process but by using VIA style network interconnects. Our system can use multiple threads per process as well, but uses different protocols to achieve this.

Other related works compare invalidation and update protocols (without RDMA usage) and program transformations to best utilize them. While the authors of [3] show some performance gains for update protocols, we avoid their extensive manual source-code transformations.

Munin [1] allows the programmer to choose from a time-out update protocol and a number of lazy release protocols. Whereas Jackal allows the protocol to be specified per object, in Munin this can be done per (global) variable. Munin's global address space is restricted. Both [3] and [1] do not use RDMA based protocol implementations and hence might benefit from the RDMA versions presented here.

3 DSM protocol template

Java's memory model prescribes that at the entry and at the exit of a synchronized block, any changes to memory caches made by a thread must be 'published' so that other threads can pick up those changes. Besides this, synchronized blocks guarantee mutual exclusion of threads. We implement the mutual exclusion part by sending a message to the owner of the object of the synchronized block and then waiting for an acknowledgement message. The owner will only send this acknowledgement if access to the synchronized block is granted. While an object has a lock associated with it, it is not eligible for home-migration (under invalidation protocols) to avoid having to migrate locking state across machines when migrating an object. Object.wait(), timed-wait(), notify(), and notifyAll() are implemented similarly.

Let us discuss the DSM protocol template first that works for both update and invalidation protocols. For concrete invalidation and update protocols different implementations of the three operations: *start_read(Region r), start_write(Region r),* and *process_cached_regions()* need to be selected. In our implementation, these functions test some flag bits in a region and then pass control to the appropriate protocol handler for that region so that protocols (invalidation or update) can be specified per region.

Our compiler inserts conditional calls to *start_read* and *start_write* into the code. We call these 'access checks'. For example, the write access to the *'p'* field at line 4 of the source code on the left hand side of Fig. 1 causes the access check in lines 11–12 to be generated. A region is already locally available if the corresponding bit in the thread's *write_bitmap* is set. Optimization passes in the compiler remove as many superfluous access checks as possible, see [10]. If the region is not yet locally available, the *start_write* and *start_read* functions cause the DSM protocol to fetch a copy of the object. After mapping the copy locally, a reference to the mapped region is added to the thread's *cached list*. On this list update and invalidation managed regions co-exist. Whereas an invalidation protocol fetches a fresh copy of a region after every single thread synchronization statement (as the use of 'synchronized' causes data invalidations), the update protocol fetches it only once whereafter it remains mapped. At

```
// Source:
class A {
    int p;
    synchronized void foo(int q) {
        this.p = q;
    }
}
```

```
// Instrumented with pseudocode:
void foo(int q) {
    process_cached_regions();
    lock(this);
    if (! current_thread.write_bitmap[this])
        start_write(this);
    this.p = q;
    process_cached_regions();
    unlock(this);
}
```

Fig. 1. DSM protocol template with object access and synchronization.

each *lock/unlock*, the *cached list* is traversed and each region is flushed (invalidation protocol) or broadcast (update protocol). In case of an invalidation protocol, a diff of a modified region is sent to the object's home-node. If the region was not modified, only a notification is sent that there is now one user less. These messages are used to implement lazy flushing, home-migration, home-only states, etc. Jackal's invalidation protocol is a multiple-writer protocol with home-migration similar to [4]. It uses lazy-flushing for cluster-wide read-only regions. In case of an update protocol, a diff is used to update all other existing copies.

As said before, Jackal uses an address translation scheme. Whenever a reference to an object is to be sent over the network, a GOR and some type-information is sent instead which the receiver maps to the local copy causing copies of objects to have different local addresses on different machines. In our prototype, the programmer can select an update protocol *per object/array* via a simple Java API. This sets the object's 'update protocol flag' and performs an all-to-all communication to exchange the local object addresses. This communication is required for RDMA protocols, as each machine needs to know the remote addresses of all the cached copies of an object. Each of *start_read/write* and *process_cached_regions()* test the object's 'update protocol flag' to determine the correct protocol handler. This all-to-all communication to exchange the addresses of local copies is needed only once at program start and is therefore not an issue for program performance. The following two sections study which parts of the general DSM protocol can/cannot benefit from RDMA.

4 Object requests by means of RDMA

Whenever an object is not locally available, the access check invokes either *start_read* or *start_write*, depending on the type of accesses that follow. The object is then requested. At receipt of the object by the requestor it is locally mapped by setting the thread's accessibility bits and by adding it to the cached-list of the requesting thread. This section shows that some parts of the object request protocol can be done by RDMA. Others must rely on regular send-receive pairs.

The fetch request itself *cannot be sent via RDMA* because of the following reasons. First, with RDMA, the home-node would have to periodically poll memory to determine message arrival. As the home-node will have other Java threads running, the program would need to be instrumented with polling statements. Due to Jackal's goal of running unchanged multi-threaded Java programs (not necessarily in SPMD style), we cannot require the Java programmers to insert polling statements in their codes. We therefore would have to resort to automatic insertion of polling statements. This causes performance problems as the frequency of polling is either too high or too low, both of which would adversely effect performance. Second, an RDMA-ed fetch request would need to correctly update any protocol state maintained at the home-mode. This would involve allocating and freeing data structures from memory, updating the accessibility state vectors, potentially sending invalidation messages, etc. These operations are too complex to manage by using only simple RDMA transfers.

We therefore need to apply a normal send-receive protocol for sending fetch-request messages. Message receipt at the home-node causes a special communication thread (an 'upcall thread') to wake up from a blocking-receive. It handles the message and sends the object back to the requestor. In contrast, object receipt by the requestor *can be handled by RDMA* as the receiver can actively wait for the message to arrive, since the message will (definitely) take only a short amount of time to arrive.

```
void fetch_request(javaObject x) {
    // determine the size of the receive area to allocate:
    int reply_msg_size = 1 + x.size() + dsm_protocol_overhead_reply(x);
    int home = x.home_node();
    // allocate a RDMA receive area from the device's buffer pool
    jackal_rdma_t *rmda_descr = jackal_rdma_alloc(home, reply_msg_size);
    if (rdma_descr) { // a suitable RDMA receive area was found,
        byte *end_msg_byte = rdma_descr->memory[reply_msg_size];
        *end_msg_byte = 0;
        send_fetch_request_with_rdma_reply(home, rdma_descr);
        // wait for the RNIC to copy the data in place
        while (*end_msg_byte == 0) {}
        process_object_reply_message(rdma_descr->local_memory);
        jackal_rdma_free(rdma_descr);
    } else {
        ack_t ack; // create a condition variable
        send_fetch_request_with_normal_reply_msg(home, &ack);
        // wait for then signal from by process_object_reply_message()
        thread_condition_wait(&ack);
    }
}
```

Fig. 2. An efficient way to use RDMA for fast object fetching.

Fig. 2 shows the pseudocode for issuing fetch requests. First, the requestor figures out how large the combined received object and its protocol data will be. If this fits in a pre-allocated and pre-registered RDMA-able memory region (jackal_rdma_alloc), the fetch request is sent with the address of the local RDMA buffer. Otherwise the object will be sent by the home-node's communication thread as a normal message. An interesting insight is that the home-node cannot directly place the object's data into the requestor's memory because in general the requestor needs to execute additional protocol code upon message receipt. For example, in a situation where there are other threads that are concurrently executing at the requestor and that already had write access to the requested object, tests need to prevent their changes from being overwritten. A naive RDMA-write initiated by the home-node cannot detect such concurrent writes as it would require it to examine the requestor's states, the (current) requestor's copy and its twin.

We therefore RDMA the complete reply message in the format of the normally sent protocol message. In other words, process_object_reply_message is always invoked for a fetch-reply message, regardless of whether the message is received normally or via RDMA. The overhead of active polling for the (RDMA-ed) message receipt is acceptable even if concurrent threads at the requestor may be slowed down that way. Note that the active memory polling for message receipt in the RDMA cannot be circumvented, nor can the CPU be freed in the meantime. We can't free the CPU using thread-yield or sleep statements as either causes slow operating system calls or takes longer than a message latency. It can be argued that polling memory in a tight loop could saturate the memory bus. Fortunately however, the reads from memory in the polling loop run out of the processor's cache which is updated by a processor's internal consistency protocols on modern CPUs.

To summerize, for fetching objects, we must send the request as a normal mesage while the reply message can be sent by RDMA. The reply message, cannot write to the object in place in order to allow multi-threaded execution. Note that our RDMA-rpc implementation is similar to what certain MPI implementations do internally for managing acks. See for example [6].

We will now examine the opportunities of RDMA use (regardless of a performance gain/loss) when we need to update copies of objects on other machines.

5 Processing the list(s) of cached objects

Each thread maintains four lists of cached objects, one for machine local read-only regions where some other machine(s) are modifying it (local read-only), one for cluster-wide read-only regions (lazy flushing), one for objects that are used by only one machine (home-only), and one for locally modified objects. At each entry and exit of a synchronized block, all regions on these lists must be examined and processed (except those on the lazy flush list). Depending on a region's flag, the region is managed by the invalidation protocol handler or by an update protocol handler. This section discusses where RDMA can be used in those protocol handlers.

5.1 Invalidation protocol handlers

For each region on the list of locally modified regions we create a diff. These diffs are then streamed to the home-node in 4 KByte packets. By streaming the diffs (instead of buffering them to send them all at once) the home-node can already process incoming diffs while the invalidator still continues to create them. Likewise, for each region on the thread's read-only list, one-user-less messages are streamed to the home-node. The region is removed from either list as soon as the messages have been sent. The home-only/lazy-flush lists are left alone. Upon receiving a diff or a one-user-less message, a state-machine quickly performs any necessary state changes to implement home migration, invalidation, or read-only replication.

Due the same reasons that prevented RDMA from being applicable to object requests (polling requirements, too complex for RDMA management, etc.), diff messages and one-user-less messages in the invalidation protocol cannot be transferred by means of RDMA either. Hence, invalidation protocols cannot exploit RDMA capabilities when propagating changes.

5.2 RDMA-based update protocol handlers

We have developed two alternative update protocols that solely use RDMA. The first one updates remote objects/arrays in place. Diffs between modified regions and their twins are created and applied (by RDMA) to the remote copies at all other machines. This is a true zero-copy protocol performing RDMA from one Java heap to another. The second update protocol stores the above diffs in a large intermediate array first, one per target machine. These arrays are then broadcast via RDMA to all other machines for local processing at their earliest convenience.

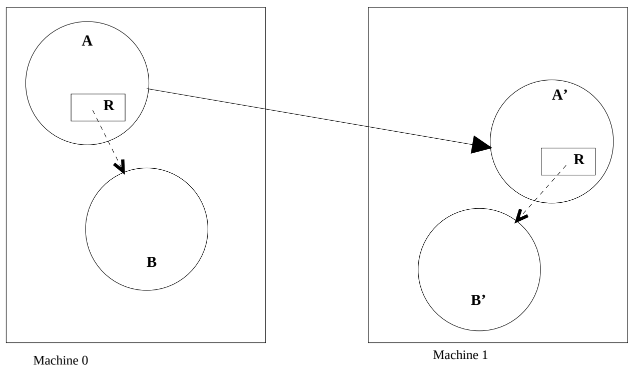

Fig. 3. Reference transfers.

For simplicity of our prototype implementation, our current update/RDMA protocols do not allow objects containing reference fields. For such objects the invalidation protocol handler must be used. To illustrate the problem, consider the example in Fig. 3. Here we have two machines, 0 and 1. Machine 0 initially holds objects A and B, where A is marked for update-protocol management. The addresses of the copies of A are therefore known at all machines. However the addresses of the copies of B at the other machines are unknown.

Now a thread at machine 0 writes a reference to B in the R field of A. This eventually causes a broadcast of A to machine 1. However, to ensure correctness, the R field of A at machine 1 should be translated to the local copy of B (and potentially allocating the copy of B if it did not already exist). A simple RDMA of A however would not do this, and write a copy of A with an illegal R field. Any circumvention of this problem would no longer make the protocol zero-copy.

To allow references in update-protocol managed objects the best solution would be to create a copy of the object in RDMA-able memory, replace references to their remotely valid equivalents, and RDMA the copy one-after-the-other to each machine. This is problematic as we need to know the remote addresses of *any* referred to objects, not only for update-protocol managed objects. This would cause memory shortage problems for maintaining the translation tables and additionally, large processing overheads as each machine would need to translate each reference for each machine to broadcast to.

Because of the difficulties outlined above, we support update protocols only for objects containing no reference fields at all (and default to an invalidation protocol for these).

Alternative 1, updating objects in place. To allow broadcasts of local modifications to objects to their copies on remote nodes by means of in-place RDMA, the entire Java object heap must be mapped and registered with the RDMA-device. Fortunately, this is not a problem with modern Infiniband hardware. Modifications to a region are found by comparison against its twin, which is a copy of the region from since it was last processed. Conceptually, for each region in the list of modified regions we invoke the update method shown in Fig. 4.

```
void update(Region r) {
    diff_t d = changes to r in respect to twin(r); apply 'd' to twin(r);
    for all machines p:
        int64_t remote_address = r->region_hash[p], remote_twin = r->twin_hash[p];
        RDMA 'd' to 'p' at remote_address and remote_twin;
}
```

Fig. 4. Update protocol handler, alternative 1: update in place.

Note that we must update both the remote object and its remote twin because of the following scenario. Assume two machines that write to an object with two fields, *X* and *Y*. One machine exclusively modifies *X* the other *Y*. Machine 0 writes to X. It then broadcasts the change to machine 1 and updates its own twin. If the other twin on machine 1 would be left untouched, the next synchronized statement (by machine 1) would cause a diff to be created for field X, causing X to be broadcast back to machine 0 overwriting any changes to X at machine 1 made in between the two broadcasts. Our solution is to update both the remote region and the remote twin.

This protocol is a zero-copy protocol, since we RDMA the changed fields from one object directly over the corresponding fields in the object copies at the other machines.

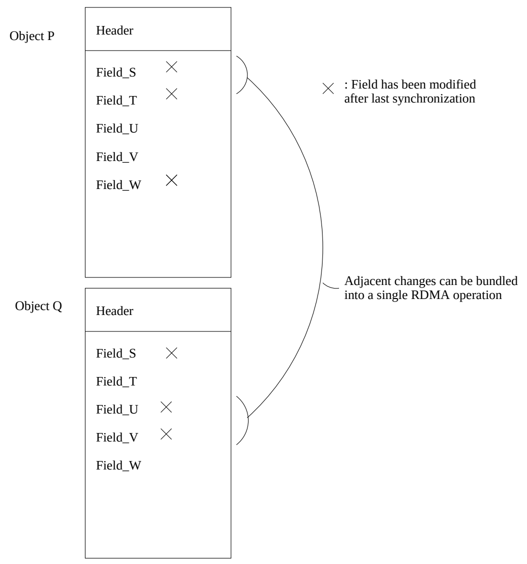
Object P
Header
Field_S
Field_T
Field_U
Field_V
Field_W
: Field has been modified after last synchronization
Adjacent changes can be bundled into a single RDMA operation
Object Q
Header
Field_S
Field_T
Field_U
Field_V
Field_W

Fig. 5. Bulk communication example.

To increase performance, we deal with ranges of fields or array elements that have changed at a time instead of processing single fields at a time. This intra-object coalescing of fields and array-element indexes allows to perform *some* bulk-RDMA. The same coalescing of changed fields inside single objects is used in Alternative 2 of the update protocol below as well.

To illustrate what exactly can and what cannot be shipped by bulk-RDMA, consider Fig. 5. Here, two objects, P and Q, have been allocated consecutively in memory. Each object is prefixed with an object header that contains information for the garbage collector, DSM, and the object's meta-information (pointer to the method table for the object, etc.). After the object-header reside the object's fields. The fields marked with an 'X' have been changed since the last lock/unlock (read: Java's synchronized).

During diff-creation, we first process object P, and create one RDMA-range for the combined fields S and T, and another RDMA-range for its field W. For object Q, we create RDMA-ranges for its field S and another for the combination of its fields U and V. To apply the diffs, we therefore have to perform four single RDMA-puts. We cannot merge the RDMA-ranges of P's W and Q's S due to the intervening object header (as the header is not read-only and furthermore, machine-local).

To summarize, this update protocol can be implemented with RDMA only. The upside of this update-protocol version is thus that there is virtually no overhead per object: (ranges of) changes in objects are copied from one machine directly to the copy on another machine. The downside is that there is no inter-object bulk-communication; only some intra-object bulk-communication is possible.

Alternative 2, RDMA-broadcast of diff-arrays. A potential problem with alternative 1 of the RDMA-update protocol is that there is little potential for bulk communication (it only supports some intra-object bulk communication). The second update protocol rectifies this by supporting inter-object bulk communication. It builds arrays of diffs and broadcasts these arrays in one go by means of RDMA, to all machines. Each machine then needs to periodically poll local memory to see if a new set of diffs has arrived and then process them. If polling is performed often enough, and if the received diff-arrays are easy to process, overheads are low. In our implementation, diff-array elements consist of a target address, a length in bytes, and the changed bytes. The methods needed to process the list of modified regions are given in Fig. 6.

For the example of Fig. 5, we would create one diff-array containing the diff-ranges {P.S-P.T}, {P.W}, {Q.S}, and {Q.U-Q.V}. These diff-ranges are then copied into RDMA-able memory and copied by one single RDMA to all other machines.

```
void process_arrived_diff_arrays() {
    for each machine m and m != myself:
        patch m.diff_array in locally;
        RDMA m.diff_array.seq_num to ’m’ at m.diff_array.remote_ack_seq_num
}
void find_locally_created_changes() {
    for all update_region r on thread->cached_modified_list:
            for each machine m: append diff(r) to m.diff_array,
                                    m.diff_array.seq_number = m.current_bcast_seq_num;
}
void broadcast_new_diff_array() {
    for each machine m:
        while m.current_bcast_seq_num != m.diff_array.acked_seq_num:
            ; // wait RDMA-ed ack
        m.current_bcast_seq_num++;
        RDMA m.diff_array to ’m’ at ’m.remote_diff_array’;
}
```

Fig. 6. Update protocol handler, alternative 2: the three steps to process diff-arrays

The diff-array protocol therefore globally performs the following three steps. We first check for diff-arrays that have arrived from other machines. Any incoming diffs are applied to the local regions and their twins. Secondly, we create local diff-arrays, one for each target machine (for regions that are not already handled by the invalidation protocol). Finally, the diff-arrays are broadcast by RDMA. Note that a shipment of a diff-array must have been acknowledged before shipping the next diff-array.

To understand the need for an ack-protocol, think of two machines 0 and 1, and two objects, *A* and *B*. Machine 0 first updates *A* and broadcasts its modifications by placing them into the diff-array in machine 1. Let us assume that directly afterwards, machine 0 were to update *B* and broadcast the changes to *B* with a new diff-array. If machine 1 had not yet acked the processing of the first diff(-array), the second broadcast would overwrite the first diff(-array). The first diff and the update to *A* would be lost. We therefore need an acknowledgment scheme.

We have implemented the acknowledgments with single-word RDMA-writes for efficiency. As soon as a machine has processed a diff-array it performs a single-word RDMA to the diff-array originator. The acknowledgement protocol itself has thus a very low overhead.

To summarize, this second update protocol can also be implemented with RDMA only. The upside of this update-protocol version is that we allow both intra- and inter-object bulk-communication to occur by sending arrays of diffs at a time. The downside of this protocol is that the receiver needs to periodically test if a diff has to be processed. If the receiver does not react quickly enough, the sender will need to wait a long time for diff-processing acknowledgements.

6 Performance

Two aspects are important for DSM performance, the latency of fetching objects and the available bandwidth for flushing or broadcasting modifications. This section analyzes performance with some micro-benchmarks and two applications. We use *Water* and *LU* from the SPLASH benchmark suite [11]. Both are irregular and challenging and thus stress the DSM protocols. Regular applications or applications with little communication are not well suited for showing protocol performance as differences are rarely visible. Also, Water and LU form the extremes of a spectrum: whereas Water uses many small objects, LU uses only one single array with larger contiguous modifications.

Our measurements were performed on a cluster of dual Xeon 3.20 GHz ”Nocona” machines (800 MHz bus, 666 MHz front-side bus) with 2 GByte RAM each. The cluster uses an Infiniband interconnect (10 GBit/s). We use our own low-latency communication package that maps directly to the Infiniband driver's libraries. Our communication package maps RDMA-puts directly on top of the Infiniband verbs layer to get the best possible performance out of Infiniband. Note that normal message sends also use the Infiniband verbs layer directly so that both normal message sends and RDMA sends are fully optimized.

Micro-benchmarks. We first evaluate Jackal's performance for some simple primitive operations so that we can eliminate these as the sources of overhead in later benchmarks. The relevant data is given in the upper part of Table 1 (to simplify presentation, we only present numbers for 1, 2, and 8 machines (read: '1, 2 and many')).

To perform a synchronized block in a loop using one or two machines costs about the same (568181 vs. 566051 locks per second). This is due to the low contention ratio for the lock. The communication costs are very low given that only very small messages are needed and most of the time these messages are handled entirely by the communication thread (the thread that handles all incoming normal messages; RDMA messages of course bypass this thread). On eight machines, the machine that holds the lock object becomes overloaded with request and release messages. Hence, performance drops. Lock contention does not yet seem to be a problem however.

Table 1. Micro-benchmark results

	1 machine	2 machines	8 machines
# Locks/second	568181	566051	1768
# Barriers/second	—	15015	9174
Object-request latency (no RDMA)	—	49.9 μs	49.9 μs
Object-request latency (RDMA)	—	24.6 μs	24.6 μs
RDMA-invalidation-flush-bandwidth	1.0 GByte/s	64.0 MByte/s	13.3 MByte/s
RDMA-in-place-update bandwidth	1.2 GByte/s	91.0 MByte/s	9.6 MByte/s
RDMA-diff-array-update bandwidth	1.2 GByte/s	84.0 MByte/s	7.7 MByte/s

For the barrier micro-benchmark we use a Java object that consists of two fields, a barrier-entry limit and a barrier-entry counter. The main work is performed in a synchronized method that increments the counter. If the counter reaches its limit, the methods calls *Object.notifyAll()*, otherwise *Object.wait()*. Each entry/exit of the synchronized method and the execution of the *wait* method causes an invalidation of the cached objects and sends the barrier object to its home.

The *lock*, *unlock*, *Object.notify()*, and *Object.wait()* are implemented as normal messages sent to the home-node of the barrier object where they attempt to locally lock the object. If the lock succeeds or the wait finishes, acknowledgement messages are sent back to the remote machine to allow it to continue. In total, including flush messages and synchronization messages, we achieve a barrier time of 109 μs per barrier on 8 machines ($\frac{1}{9174}$). While the absolute number may seem high, given the much lower latencies for, say, an MPI barrier on Infiniband, Java's semantics add significant overhead to a barrier. For example, to handle multi-threading, Java requires an implementation to flush working memory, to send synchronization related messages, to handle Object.wait() and Object.notify(), and finally to wake up threads from thread-pools to handle protocol messages. With all this overhead, 109 μs is quite good.

Object request speed is measured by traversing a linked list. Each access to the 'next' field in a linked node causes that node to be fetched. For a list containing N elements, we thus get 2 N messages (one request message, one reply message with the node's data). We then take the average time required for a single list node fetch. Enabling RDMA for object requests almost halves the latency for fetching a node (even though only one side of the round trip can be optimized via RDMA). The numbers include the time needed for state updates, address translations, and for allocating a local copy for every list node. Regardless of whether 2 or 8 machines are used, the times are the same due to the low protocol processing overheads.

User level bandwidth is measured as follows. All machines (1 thread per machine) concurrently execute a loop and change each N-th element of a 32KByte array. The modifications are propagated via a single, empty synchronized block. This inner loop is performed 20.000 times to give us an indication of the application-level bandwidth available (32K * 20.000 / # seconds used). The same benchmark is used for the invalidation protocol and the two update protocols. When only 1 or 2 machines are used, both update protocols outperform the invalidation protocol. With larger numbers of machines, the invalidation protocol wins due to the high overheads in both update protocols. Note that for all bandwidth measurements, RDMA is also used for region requests (repeatedly for the invalidation protocol, once for the update protocols).

```
class MolData {
    double [][]data = new double [NUM_DIMENSIONS] [NUM_ATOMS];
    ...
}

class MoleculeEnsemble {
    MolData[][] f = new MolData [MAX_ORDERS] [ getNumMolecules() ];
    ...
}
```

Fig. 7. Water's main datastructures.

Water performs an (N-square) N-body simulation of water molecules coded as in Fig. 7. We simulate only 1728 molecules to stress protocols. The innermost array elements (the NUM_ATOMS dimension of MolData), contain the actual molecule data. The other data here is read-only and is cluster-wide read-only replicated by the protocols. Note that NUM_ATOMS equals '3' here (for two hydrogen atoms and one oxygen atom). This stresses protocols as modified data is encapsulated in many of these small arrays.

An invalidation protocol with switched on RDMA-request, improves performance by 20.6% on 8 machines (13.1 seconds with RDMA-request versus 16.5 seconds with regular messaging, see Table 2).

Regardless of the number of machines used, both update protocols are slower as we pay two penalties. First, although changes are broadcast to every machine (eliminating the need to explicitly fetch them), the changes are not used by *every* processor. The exact set of consumers is hard to detect by the DSM protocol without changing Water's source code. In contrast, the invalidation protocol pulls the changes to only those machines that require them.

The second penalty is due to Java's lack of true multi-dimensional arrays (Java provides only arrays of references to sub-arrays). Since the data of all the innermost 1D arrays are not contiguous in memory, the first RDMA update protocol (updates in place) needs a large number of RDMA transfers. On the other hand, the diff-array RDMA update protocol version has a lot of administrative overhead for each diff-array. Even the higher efficiency of the RDMA hardware (compared to normal send/receive) cannot overcome this. The slight advantage of diff-array RDMA-update protocol on only two machines is quickly lost with increasing numbers of machines.

Table 2. Application results in walltime (seconds)

	1 machine	2 machines	8 machines
Water, no RDMA	56.4	41.2	16.5
Water, RDMA-request invalidation	56.4	40.2	13.1
Water, RDMA-in-place-update	56.9	41.1	26.8
Water, RDMA-diff-array-update	56.9	36.6	20.0
LU, no RDMA	47.3	30.1	19.6
LU, RDMA-request invalidation	47.3	32.9	18.9
LU, RDMA-in-place-update	47.0	37.5	26.1
LU, RDMA-diff-array-update	47.0	41.1	23.2

LU factorizes a dense matrix, encoded as a single flattened array of doubles. Due to the blocking technique used, every machine accesses only a few linear segments of array elements. Hence the number of region fetches needed is less than in Water. Use of RDMA improves the performance of the invalidation protocol only slightly by 3.6% (19.6 versus 18.9 seconds).

The update protocols are always a loss for LU. Unlike Water, LU's threads write larger consecutive chunks in a single linearized matrix. Because of array chunking, fewer updates and hence fewer RDMA broadcasts are needed. A few hundreds of 2 KByte array segments are broadcast instead of tens of thousands of 24 byte broadcasts as in Water. LU also suffers from the effect that broadcast data is often not (immediately) used by the receiving CPUs. Hence the invalidation protocol (with RDMA fetch) is faster.

Finally, there are many array sections on the list of modified regions that are actually *unmodified* since the last list processing. This happens because in update protocols, modified regions are never removed from the modified-regions list, causing many empty diffs. However, since we still need to test every single array element for potential modifications at each synchronization, the processing requirements of the update protocols increase. In Water, this effect does not occur since each processor writes the same water molecules each time. Of course, the invalidation protocol does not suffer from this effect as its regions are removed from the modified-regions list at invalidation time (but each access afterward triggers an access check to add it again to one of the flush-lists).

7 Conclusions

In our system, invalidation and update protocol managed regions can coexist. We found that RDMA can be successfully applied to invalidation protocols and have designed two update protocols that solely use RDMA. We have seen performance inprovements of up to 20.6% using RDMA for object-fetching. Without source code changes (for example, those suggested by [3]), even when modern RDMA hardware is used throughout, update protocols are still slower than invalidation protocols. This is because of three main reasons. First, when adding address translation to allow large address spaces, the cost of protocol processing grows large in update-protocols as they need, per-machine processing.

Second, message aggregation is hard to do with current Infiniband RDMA implementations as they currently lack remote scatter. Ideally, we would like to present the RDMA hardware with two lists of I/O vectors. One I/O vector for where to copy the data from at the local machine, and another I/O vector for where to copy the data to at the target machine. The current Infiniband VERBS allows only very limited use of I/O vectors.

Finally, another RDMA-feature currently missing is signalled-IO. Signalled RDMA would cause an interrupt at the receiver once data has been copied. This would not only allow to free the CPU when waiting for message replies but it would also allow us to immediately reply to unexpected incoming messages (instead of periodically tested for them).

References

1. J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, 1995.
2. H. Eichner, C. Trinitis, and T. Klug. Implementation of a DSM-System on Top of InfiniBand. In *Proc. 14th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 178–183, Washington, DC., Feb. 2006.
3. B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J. R. Larus, A.Rogers, and D.A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, pages 380–389, Washington, DC, Nov. 1994.
4. W. Fang, C.L. Wang, W. Zhu, and F.C.M. Lau. A novel adaptive home migration protocol in home-based DSM. In *Proc. of the 2004 IEEE Intl. Conf. on Cluster Computing*, pages 215–224, San Diego, CA, Sep. 2004.
5. V. Iosevich and A. Schuster. Multithreaded Home-Based Lazy Release Consistency over VIA. In *Proc. 19th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'04)*, pages 59–70, Santa Fe, New Mexico, Apr. 2004.
6. J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS 04)*, Santa Fe, NM., Apr 2004.
7. R. Noronha and D. K. Panda. Reducing Diff Overhead in Software DSM Systems using RDMA Operations in InfiniBand. In *Workshop on Remote Direct Memory Access (RDMA): RAIT 2004, (Cluster '04)*, San Diego, CA, Sep. 2004.
8. J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proc. 28th Hawaii Intl. Conf. on System Sciences (HICSS'95)*, pages 74 – 84, Jan. 1995.
9. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang and H.E. Bal. Runtime-Optimizations for a Java DSM. In *Proc. ACM 2001 Java Grande Conf.*, pages 89–98, San Francisco, CA, June 2001.
10. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, C.J.H. Jacobs, and H.E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *8th Symp. on Principles and Practices of Parallel Programming (PPoPP)*, pages 83–92, Snowbird, Utah, June 2001.
11. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.