The Remote Enqueue Operation on Networks of Workstations

Evangelos P. Markatos and Manolis G.H. Katevenis and Penny Vatsolaki*

Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH), Crete
P.O.Box 1385 Heraklio, Crete, GR-711-10 GREECE
markatos@ics.forth.gr
http://www.ics.forth.gr/proj/avg/telegraphos.html

Appears in Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC 98), Las Vegas, USA, 1998

Abstract. Modern networks of workstations connected by Gigabit networks have the ability to run high-performance computing applications at a reasonable performance, but at a significantly lower cost. The performance of these applications is usually dominated by their efficiency of the underlying communication mechanisms. However, efficient communication requires that not only messages themselves are sent fast, but also notification about message arrival should be fast as well. For example, a message that has arrived at its destination is worthless until the recipient is alerted to the message arrival.
In this paper we describe a new operation, the *remote-enqueue* atomic operation, which can be used in multiprocessors, and workstation clusters. This operation atomically inserts a data element in a queue that physically resides in a remote processor's memory. This operation can be used for fast notification of message arrival, and for fast passing of small messages. Compared to other software and hardware queueing alternatives, remote-enqueue provides high speed at a low implementation cost without compromising protection in a general-purpose computing environment.

1 Introduction

Popular contemporary computing environments are comprised of powerful workstations connected via a network which, in many cases, may have a high throughput, giving rise to systems called *workstation clusters* or Networks of Workstations (NOWs) [1]. The availability of such computing and communication power gives rise to new applications like multimedia, high performance scientific computing, real-time applications, engineering design and simulation, and so on. Up to recently, only high performance parallel processors and supercomputers were

* The authors are also with the University of Crete.

able to satisfy the computing requirements that these applications need. Fortunately, modern networks of workstations connected by Gigabit networks have the ability to run most applications that run on supercomputers, at a reasonable performance, but at a significantly lower cost. This is because most modern Gigabit interconnection networks provide both low latency and high throughput. However, efficient communication requires that not only messages themselves are sent fast, but also notification about message arrival should be fast as well. For example, a message that has arrived at its destination is worthless until the recipient is alerted to the message arrival.

In this paper we present the *Remote Enqueue* atomic operation, which allows user-level processes to enqueue (short) data in remote queues that reside in various workstations in a cluster, with no need for prior synchronization. This operation was developed within the Telegraphos project [18], in order to provide a fast message arrival notification mechanism. The Telegraphos network interface provides user applications with the ability to read/write remote memory locations, using regular load/store instructions to remote memory addresses. Sending (short) messages in Telegraphos can be done by issuing one or more remote write operation, which eliminates traditional operating system overheads that used to dominate message passing. Thus, sending (short) messages can be done from user-level by issuing a few store assembly instructions. Although sending a message can be done fast, notifying the recipient of the message arrival may take significant overhead. For example, one might use a shared flag in which the sender writes the memory location (in the recipient's memory) where the message was written. When the recipient checks for messages, it reads this shared flag and finds out if there is an arrived message and where it is. However, if two or more senders attempt to send a message at about the same time, only one of them will manage to update the flag, and the other's update will be lost. A solution would be to have a separate flag for each possible sender. However, if there are several potential senders, this solution may result in significant overhead for the receiver, who would be required to poll too many flags. Arranging the flags in hierarchical (scalable) data structures might reduce the polling overhead, but it would increase the message notification arrival overhead.

Our solution to the message arrival notification problem is to create a remote queue of message arrival notifications. A remote queue is a data structure that resides in the remote node's main memory. After writing their message to the receiver's main memory, senders enqueue their message arrival notifications in the remote queue. Receivers poll their notification queues to learn about arrived messages. Although enqueueing notifications in remote queues can be done completely in software, we propose a hardware *remote enqueue* operation that *atomically* enqueues a message notification in a remote queue. The benefits of our approach are:

- *Atomicity at low cost*: to prevent race conditions, all software-implemented enqueue operations are based on locking (or on `fetch_and_`ϕ) operations that appropriately serialize concurrent accesses to the queue. These operations incur the overhead of at least one network round-trip delay. Our hardware-implemented remote enqueue operation serializes concurrent enqueue operations at the receiver's network interface, alleviating the need for round-trip messages.

– *Low-latency flow control:* Most software-implemented enqueue operations may delay (block) the enqueing process if the queue is full. For this reason, most software-implemented enqueue operations need to read some metadata associated with the remote queue in order to make sure that the remote queue is not full. Unfortunately, reading remote data may take at least one round-trip network delay. In our approach, the enqueueing process *always* succeeds; if the queue fills up after an enqueue operation, a software handler is invoked (at the remote node) to allocate more space for the queue. Since our remote enqueue operation is non-blocking, and does not need to read remote data, it can return control to its calling processes, as soon as the data to be enqueued have been entered in the sender's network interface, that is the remote enqueue operation may return control within a few (network interface) clock cycles - usually a fraction of a microsecond.

The rest of the paper is organized as follows: Section 2 surveys previous work. Section 3 presents a summary of the Telegraphos workstation cluster. Section 4 presents the remote enqueue operation, and section 5 summarizes this paper.

2 Related Work

Although networks of workstations may have an (aggregate) computing power comparable to that of supercomputers (while costing significantly less), they have rarely been used to support high-performance computing, because communication on them has traditionally been very expensive. There have been several projects to provide efficient communication primitives in networks of workstations via a combination of hardware and software: Dolphin's SCI interface [19], PRAM [24], Memory Channel [13], Myrinet [6], ServerNet [26], Active Messages [12], Fast Messages [17], Galactica Net [16], Hamlyn [9], U-Net [27], NOW [1], Parastation [28], StarT Jt [15], Avalanche [10], Panda [2], and SHRIMP [4] provide efficient message passing on networks of workstations based on memory-mapped interfaces. We view our work as complimentary to these projects, in the sense that we propose a fast message notification mechanism that can improve the performance of all these message passing systems.

Brewer *et. al* proposed *Remote Queues*, a communication model that is based on enqueueing and dequeuing information in queues in remote processors [8]. Although their model is mostly software based, it can be tuned to exploit any existing hardware mechanisms (e.g. hardware queues) that may exist in a parallel machine. Although their work is related to ours we see two major differences:

– *Remote queues* combine message transfer with message notification: the message itself is enqueued in the remote queue. The receiver reads the message from the queue and (if appropriate) copies the message to its final destination in its local memory. In our approach we assume that the message has

been posted directly in its final destination in the receiver's memory, and only the notification of the message arrival need to be put in the queue - our approach results in less message copy operations. Suppose for example that the sender and the receiver share a common data structure (e.g. a graph). Using out approach, the sender deposits its information directly in the remote graph, where the receiver will read it from. On the contrary, in the remote queues approach, the messages are first placed in a queue, and the receiver will have to copy the messages from the queue and put their information on the common graph, resulting in one extra copy operation. Recent commercial network interfaces like the Memory Channel and the PCI-SCI efficiently support our approach of the direct deposit of data in the receiver's memory.
- *Remote Queues* have been designed and implemented in commercial and experimental massively parallel processors that run parallel applications in a controlled environment, supporting little or no multiprogramming. Our approach has been designed for low-cost Networks of Workstations that support sequential and parallel applications at the same time.

In single-address-space multiprocessors, our remote enqueue operation can be completely implemented in software using any standard queue library. Brewer *et. al* propose such an implementation on top of the Cray T3D shared-memory multiprocessor [8]. Any such implementation (including the one in [8]) suffers from software overhead that includes at least one atomic operation (to atomically get an empty slot in the queue), plus several remote memory accesses (to place the data in the remote queue and update the remote pointers). This overhead is bound to be significant in a Network of Workstations.

In many multiprocessors, nodes have a network co-processor. Then, the remote enqueue operation can be implemented with the help of this co-processor. The co-processor implements sophisticated forms of communication with the processes running on the host processor. For example, a process that wants to enqueue a message in a remote queue, sends the message to the co-processor, which forwards it to the co-processor in the remote node, which in turn places the message in the remote queue. Although the existence co-processors improves the communication abilities of a node, it may result (i) in software overhead (after all they are regular microprocessors executing a software protocol), and (ii) in increased end-system cost.

3 The Telegraphos NOW

The Remote enqueue operation described in this paper is developed within the Telegraphos project [22]. *Telegraphos* is a distributed system that consists of network interfaces and switches for efficient support of parallel and distributed applications on a workstation cluster. We call this project Telegraphos or $T\eta\lambda\acute{\epsilon}\gamma\rho\alpha\phi o\varsigma$ from the greek words $T\eta\lambda\acute{\epsilon}$ meaning remote, and $\gamma\rho\acute{\alpha}\phi\omega$ meaning write, because the *central* operation on Telegraphos is the remote write operation. A remote write operation is triggered by a simple `store` assembly instruction, whose argument is a (virtual) memory address mapped on the physical memory of another workstation. The remote write operation makes possible the (user-to-user, fully protected) sending of short messages with a *single* instruction. For comparison, traditional workstation clusters connected via FDDI and ATM take several thousands of instructions to send even the shortest message across the network. Telegraphos also provides remote read operations, DMA operations, atomic operations (like `fetch_and_increment`) on remote memory locations, and a non-blocking `fetch(remote,local)` operation that copies a remote memory location into a local one. Finally, Telegraphos also provides an eager-update multicast mechanism which can be used to support both multicasted message-passing, and update-based coherent shared memory.

Telegraphos provides a variety of hardware primitives which, when combined with appropriate software will result in efficient support for shared-memory applications. These primitives include:

- *Single remote memory access*: On a *remote* memory access, traditional systems require the help of the operating system, which either replicates locally the remote page and makes a local memory access, or makes the single remote access on behalf of the requesting process. To avoid this operating system overhead, Telegraphos provides the processor with the ability to make a read or write operation to a remote memory location without replicating the page locally and without any software intervention; just like shared-memory multiprocessors do [3].
- *Access counters:* If a page is accessed by a processor frequently, it may be worthwhile to replicate the page and make all accesses to it locally. To allow informed decisions, Telegraphos provides access counters for each remotely-mapped page. Each time the processor accesses a remote page, the counter is decremented, and when it reaches zero an interrupt is sent to the processor which should probably replicate the page locally [7, 20, 21].
- *Hardware multicasting:* Telegraphos provides a write multicast mechanism in hardware which can be used to implement one-to-many message passing operations, as well as an update-based memory coherence protocol. This multicast mechanism uses a novel memory coherency protocol that makes sure that even when several processors try to update the same data and multicast their updates at the same time, they will all see a consistent view of the updated data; details about the protocol can be found at [22].
- *User-level DMA:* To facilitate efficient message passing, Telegraphos allows user-level initiation of all shared-memory operations including DMA. Thus, Telegraphos does not need the involvement of the Operating System to transfer information from one workstation to another [23].

The Telegraphos network interface has been prototyped using FPGA's; it plugs into the TurboChannel I/O bus of DEC Alpha 3000 model 300 (Pelican) workstations.

4 Remote Enqueue

We propose a new atomic operation, the *remote enqueue (REQ)* atomic operation. The REQ atomic operation is invoked with two arguments:,

- *REQ(vaddr,data)*, where *vaddr* is the virtual address that uniquely identifies a remote queue (a remote queue always resides on the physical memory of a different processor from the one invoking the REQ operation), and *data* is a single word of information to be inserted in the queue. This information is most usually a virtual address (pointer) that identifies the message body that the processor invoking the REQ operation has just sent to the processor that hosts the queue in its memory. [2].

We define a *remote queue* to be a portion of a remote processor's memory that is managed as a FIFO queue. This FIFO queue is a linked list of buffers which are physically allocated in the remote processor's memory. Data are placed in this FIFO queue by the *remote enqueue (REQ)* operation, implemented in hardware. Data are removed from this FIFO queue with a *dequeue* operation which is implemented in user-level software.

The following limitations are imposed to the buffers of a remote queue, for the hardware remote enqueue operation to be efficient:

- The starting address of each buffer should be an integer multiple of the buffer size, which is a power of two.
- The maximum buffer size is 64KB (for a 32-bit word processor).
- If the buffer size is larger than the page size, each buffer should be allocated in contiguous physical pages.

The layout of the buffer is shown in figure 1. The *head* and *tail* are indices in the *data buffer*. A queue is a linked list of such data buffers. For a 32-bit-word processor, both *tail*, and *head* are 16 bit quantities, and not full memory addresses. The reason is that, in traditional systems (where *tail* and *head* are full addresses), we calculate the pointer to the head (or the tail) of the queue by adding *addr* with *head* (or *tail*), meaning that we need to pay the hardware cost of an extra adder, and the performance cost of a word-length addition. In our system instead, where the *addr* is a multiple of a power of two, and both *head* and *tail* are always less than this power of two, we calculate the pointer to the head (or the tail) of the queue by performing an inexpensive OR operation instead of an expensive addition.

4.1 The Enqueue Operation

When processor A wants to enqueue some *data* in the remote queue *vaddr* that physically resides on processor B's memory, it invokes the *REQ(vaddr,data)*

[2] The Telegraphos network always delivers remote data *in-order* from a given source to a given destination node. Thus, data can never arrive before the corresponding REQ operation is posted

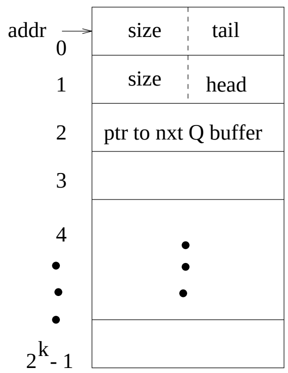

Fig. 1. Layout of a data buffer. A remote queue is just a linked list of such buffers. The first three words of the buffer are reserved to store the size, the tail, the head, and the pointer to the next Q buffer.

atomic operation. A portion of this operation in implemented on the sender node's network interface, and another portion of this operation is implemented on the receiver node's network interface.

The Sender Node: When the software issues a REQ(vaddr,data) atomic operation, the local network interface takes the following actions:

- It prepares a *remote-enqueue-request* packet to be sent to the remote node that contains *paddr* (the physical address that corresponds to virtual address *vaddr*), and *data*, and
- It releases the issuing processor, which is able to continue with the rest of its program, without having to wait for the remote enqueue operation to complete.

The Receiver Node: When the destination node receives a *remote-enqueue-request* packet it extracts the *paddr* and *data* arguments from the packet and performs the remote-enqueue operation as the following atomic sequence of steps:

- Writes the *data* to the buffer entry pointed by the *tail* index (the address of the entry is calculated as *(paddr OR tail)*.
- Increments the *tail* by 1 modulo buffer *size* (If the *tail* equals the *size* of the buffer, then *tail* gets the value of the first available buffer location: 3 (see figure 1)).
- If the buffer overflows ($tail = head$), the network interface stops accepting incoming network requests, and sends an interrupt to the (destination) processor.

The hardware finite state machine (FSM) of the destination HIB for the remote enqueue operation "req(addr, data)" is shown in table 1.

```
1.   read  (addr)        -> (size, tail) // read tail and size of Q
        2.   write (addr OR tail)<- (data)  // insert new element in Q
             // Note: (addr OR tail) points to the first free element in the Q
             // thus: no adder is needed
        3.   tmp                 <- (tail + 1) // increment tail modulo size
             // if (tmp == size) then tail = 3
        4.   if (tmp & size)     then
                   tail <- 3
             else
                   tail <- tmp
             // Note: if (tmp == size) then (tmp & size) == 1
             // else  (tmp & size) == 0,
             // thus the comparison can be implemented with AND
             // gates instead of a general purpose comparator
        5.   read  (addr+1)      -> (size,head) // read head of Q
        6.   if (head == tail) then
                   stop_accepting_network_requests()
                   interrupt host (overflow)
             else
                   write (addr)  <- (size,tail)
```

Table 1. Finite State Machine for the Remote Enqueue Operation.

Hardware Diagram: The Telegraphos datapath for the remote enqueue operation (at the receiver side) is shown in figure 2. The whole operation is controlled by five control signals: LD0, RD0, WR0, RD1, and WR1, that are generated by a simple Finite State Machine in the above order.

- LD0 loads the ADDRESS and DATA registers with the address and data that are the arguments of the remote enqueue operation.
- RD0 starts the reading of the $(size, tail)$ pair from $address$.
- WR0 starts the writing of the $data$ into the remote queue at address $addr\ OR\ tail$
- RD1 starts the reading of the $(size, head)$ pair from $(address + 1)$.
- Finally, WR1 writes the new $(size, tail)$ pair into $address$

4.2 Handling Buffer Overflow

When the current buffer fills up, an interrupt is sent to the processor which starts executing the operating system. The actions that the operating system should take are:

- Copy the contents of the full buffer into an empty one. Mark the previously overflowed buffer as empty.
- Link the new buffer into a queue of buffers associated with this queue. The *next* field in the queue is used for this purpose.
- Enable the Network Interface to handle all requests.

4.3 Dequeuing and Queue Handling in the Receiver Software

In this section we outline how the dequeue operation can be efficiently implemented in software at user-level. A straightforward implementation of the dequeue operation would be:

```
deq(queue)
{
     buffer = find_last_buffer_following_the_next_pointers() ;
     if (is_empty(buffer) {
        if (is_first(buffer, queue))
            return EMPTY_QUEUE ;
        else  {
            deallocate(buffer) ;
            buffer = find_last_buffer_following_the_next_pointers() ;
        }
     }
     result =  buffer[head] ;  head ++ ;
     if (head == size)
         head = 3 ;
     return result
}
```

Unfortunately, the above solution does not always work, because it is executed in user-space, and as such, it may be interrupted at any time. For example, consider the following scenario:

- A dequeue operation starts executing, taking an element from the head buffer (say A) of the queue.
- Before the operation completes, it is interrupted.
- In the meanwhile, the head buffer overflows, the operating system takes control, copies the buffer A into an empty one (say B), resetting the previously full buffer A.
- Some more remote enqueue operations are executed, completely overwriting the previous data on A (which are safely copied into the recently allocated buffer B).
- The dequeue operation eventually resumes execution trying to dequeue elements from buffer A, which does not have the elements the dequeue operation expects to find, which are now in buffer B!

Fortunately, on the Alpha processor there is a special mode the *PAL mode* which enables (super) users to write their own code (of limited size) and run it uninterrupted [25]. Thus, if the above code is turned into PAL code, it will run uninterrupted. PAL code is invoked via the special *cal_pal* routine, that the DEC Alpha processor provides. Although any user is allowed to call a PAL function, only the super user is allowed to install new PAL functions, thereby protecting the integrity of the system. Thus, the above mentioned race conditions disappear because the dequeue operation runs uninterrupted in PAL mode.

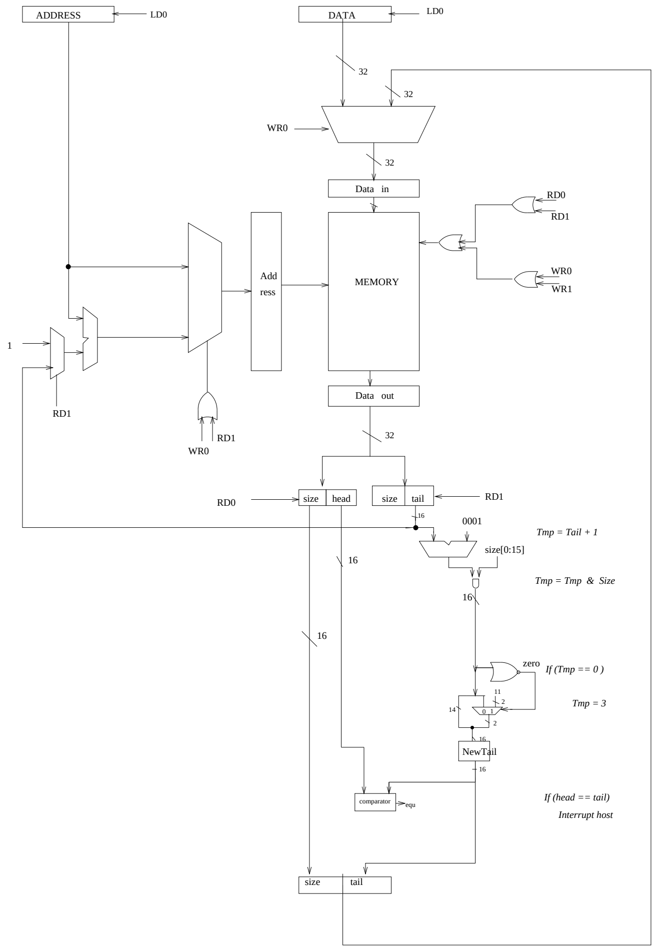

Fig. 2. The enqueue hardware.

Although PAL calls are an elegant way of executing short sequences of instructions uninterrupted, they are specific to the Alpha processor. Moreover, interrupt disabling (and of course PAL calls) is an effective way of synchronization only in uniprocessors. Disabling interrupts in symmetric multiprocessing systems that share a common network interface does not necessarily guarantee the absence of race conditions. For this reason, we have developed a more general solution that allows dequeue operations to proceed at user-level without the need to invoke PAL calls. Our solution is based on the collaboration between the operating system and the library that implements the dequeue operation. We assume the existence of a "do-not-preempt-me" bit (per queue) that is shared by the user application and the kernel. [3] When the application is about to execute a dequeue operation, it sets the "do-not-preempt-me" bit. When the dequeue operation completes, it resets the "do-not-preempt-me" bit. If the queue becomes full while an application is dequeuing something from the queue, the operating system driver that handles the buffer overflow interrupt, does not allocate a new buffer but sets a "full-queue" flag. When the interrupt handler returns, the application will resume execution, and it will complete the dequeue operation. When the dequeue operation completes, it checks the "full-queue" flag. If the flag is set, the application will invoke the network interface driver (e.g. through an `ioctl` call) to allocate more space for the queue and to enable the network interface to handle further enqueue operations. This solution works even in multiprocessor workstations that share a single network interface, with only one additional requirement: threads that execute concurrent dequeue operations (from the same queue) have to synchronize through a lock variable (associated with the queue). The first instruction of a dequeue operation is to acquire the lock, and the last instruction is to release the lock. Thus, while a thread is dequeuing data from a queue, no other thread is allowed to do the same, and thus no other thread can access shared information like the "do-not-preempt-me" bit and the "full-queue" flag. In case of buffer overflow, user-level threads should keep the lock up till the time the operating system allocates more space for the queue. If the queue fills up while at the same time a thread is executing a dequeue operation, the operating system allows the dequeue operation to complete; after the operation completes it invokes the operating system to allocate more space for the queue and to enable further network transactions.

4.4 Issuing an Enqueue Operation

An enqueue operation is invoked as: *enq(vaddress,data)* (where *vaddress* is the virtual address of the base of the first queue buffer and *data* are the data to be enqueued). In order to create a valid *remote enqueue request* packet, the network interface needs to know the *physical* address *paddr* that corresponds to virtual address *vaddr*, as well as the *data* argument. However, users are not allowed to communicate physical addresses to the network interface, because (i) they no dot know the mapping between virtual and physical pages, and (ii) malicious or ignorant users may request enqueue operations to physical addresses on which they do not have read/write access. To alleviate this problem we use the mechanism of *shadow-addressing* [5, 14, 23]. The method of shadow addressing is used to securely translate virtual to physical addresses and pass them to the network interface from user-level processes. For each virtual address `vaddr` that is mapped in the physical address `paddr`, there is also a shadow address `shadow(vaddr)`, which is mapped in the shadow physical address `shadow(paddr)`.[4] The shadow

[3] Similar mechanisms has been used to avoid preempting a user-level thread while executing in a critical section [11].

[4] The Operating System is responsible for creating both mappings at memory allocation (initialization) time.

function is simple and known to the network interface. One simple shadow function is to concatenate each address with an extra shadow bit. When the shadow bit is set, then the address is a shadow one. For example, 0x0FFFFFFFF is a regular 33-bit address, while 0x1FFFFFFFF is its shadow address.

An access to a shadow address is always interpreted by the network interface as a special argument passing operation. For example, suppose that virtual address `vaddr` is mapped to physical address `paddr`, and that the virtual address `shadow(vaddr)` is mapped into `shadow(paddr)`. Normally, a load (store) operation to virtual address `vaddr` by a user application is translated by the TLB (page-table) into a load (store) operation to physical address `paddr` and is performed by the appropriate memory controller. Similarly, a load (store) operation to virtual address `shadow(vaddr)` is translated by the TLB into a load (store) operation to physical address `shadow(paddr)`. When, however, this operation reaches the network interface it will be treated as an argument passing operation, and neither a load nor a store operation will be performed to physical address `shadow(paddr)`. Thus, when the user application wants to pass to the network interface the physical address `paddr`, it makes a store operation to virtual address `shadow(vaddr)`. Eventually the physical address `shadow(paddr)` reaches the network interface, which recognizes the shadow address and takes the physical address `paddr` by applying function $\texttt{shadow}^{-1}$ to physical address `shadow(paddr)`. [5]

Thus, a remote enqueue atomic operation is issued using a single assembly instruction as follows:

REQ (vaddr, data)
    / pass physical address shadow(paddr) to the*
    *** network interface */*
    STORE data **TO** shadow(vaddr)

5 Summary

In this paper we describe a new operation, the *remote-enqueue* atomic operation, which can be used in multiprocessors, and workstation clusters. This operation atomically inserts a data element in a queue that physically resides in a remote processor's memory. This operation can be used for fast notification of message arrival, and for fast passing of small messages. Both enqueue and dequeue operations can be issued from user-level processes without any need to call the operating system. Both operations enforce standard virtual memory protection when accessing remote queues, and thus they provide full protection in a general-purposed multiprogrammed environment. Compared to other software and hardware queueing alternatives, remote-enqueue provides high speed at a

[5] All shadow addresses should be within the physical address range of the network interface, and distinct from the normal physical addresses used by that network interface.

low implementation cost without compromising protection in a general-purpose computing environment.

Acknowledgments

This work was supported in part by ESPRIT project 6253 "Supercomputer Highly Parallel System" (SHIPS), funded by the European Union, through DG III of its Commission, HPCN Unit. We deeply appreciate this financial support, without which this work would have not existed. A patent application for the above work has been filed: E. Markatos, M Katevenis, and P. Vatsolaki: "Notification of message arrival in a parallel computer system", Patent application number 97410036.4, (Europe) March 19th 1997.

References

1. T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.
2. H. Bal, R. Hofman, and K. Verstoep. A Comparison of Three High Speed Networks for Parallel Cluster Computing. In *Proc. 1st International Workshop on Communication and Arch. Support for Network-Based Parallel Computing*, pages 184–197, 1997.
3. BBN Advanced Computers Inc. *Inside the TC2000^{TM} Computer.* Cambridge, Massachusetts, February 1990.
4. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 142–153, Chicago, IL, April 1994.
5. M.A. Blumrich, C.Dubnicki, E.W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture*, pages 154–165, San Jose, CA, February 1996.
6. N.J. Boden, D. Cohen, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29, February 1995.
7. William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.
8. E.A. Brewer, F.T. Chong, L.Tl Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Symp. on Parallel Algorithms and Architecures*, 1995.
9. G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: a High-performance Network Interface, with Sender-Based Memory Management. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.
10. A. Davis, M. Swanson, and M. Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. Technical report, University of Utah, Dept. of Computer Science, 1996.

11. J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. Technical Report Ultracomputer Note 136, Ultracomputer Research Laboratory, New York University, April 1988.
12. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19-th International Symposium on Comp. Arch.*, pages 256–266, Gold Coast, Australia, May 1992.
13. R. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12, February 1996.
14. J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1994.
15. James C. Hoe and Mike Ehrlich. StarT-JR: A Parallel System from Commodity Technology. In *Proceedings of the 7th Transputer/Occam International Conference*, November 1995. Tokyo, Japan.
16. Andrew W. Wilson Jr., Richard P. LaRowe Jr., and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proc. 13-th Int. Conf. on Distr. Comp. Syst.*, pages 246–255, Pittsburgh, PA, May 1993.
17. V. Karamcheti, S. Pakin, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing 95*, 1995.
18. Manolis G. H. Katevenis, Evangelos P. Markatos, George Kalokerinos, and Apostolos Dollas. Telegraphos: A Substrate for High-Performance Computing on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 43(2):94–108, June 1997.
19. O. Lysne, S. Gjessing, and K. Lochsen. Running the SCI Protocol over HIC Networks. In *Proceedings of the Second International Workshop on SCI-based Low-cost/High-perfocmance Computing (SCIzzL-2)*, March 1995. Santa Barbara, CA.
20. E.P. Markatos. Using Remote Memory to avoid Disk Thrashing: A Simulation Study. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, pages 69–73, February 1996.
21. E.P. Markatos and C.E. Chronaki. Trace-Driven Simulations of Data-Alignment and Other Factors affecting Update and Invalidate Based Coherent Memory. In *Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 44–52, January 1994.
22. E.P. Markatos and M. G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture*, pages 144–153, Feb 1996. URL: http://www.csi.forth.gr/ proj/arch-vlsi/papers/ 1996.HPCA96.Telegraphos.ps.gz.
23. E.P. Markatos and M. G.H. Katevenis. User-Level DMA without Operating System Kernel Modification. In *Proc. of the 3rd International Symposium on High Performance Computer Architecture*, pages 322–331, Feb 1997. URL: http://www.csi.forth.gr/proj/aavg/papers/ 1997.HPCA97.user_level_dma.ps.gz.
24. D. Serpanos. *Scalable Shared-Memory Interconnections.* PhD thesis, Princeton University, Dept. of Computer Science, October 1990.
25. R. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

26. Tandem Computers Inc. ServerNet Technology: Introducing the Worlds First System Area Network, 1996. http://www.tandem.com/INFOCTR/BRFS_WPS/SNTSANWP/SNTSANWP.HTM.
27. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 15-th Symposium on Operating Systems Principles*, pages 40–53, December 1995.
28. Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaPC / ParaStation Project: Efficient Parallel Computing by Clustering Workstations. Technical Report 13/96, University of Karlsruhe, Dept. of Informatics, 1996.