

Fast Communication in Distributed Systems Using the Networked Virtual Memory System

Max Morris, Akira Jinzaki, Tadafusa Niinomi,
Yuzo Ageno*, Shinji Kobayashi

Fujitsu Laboratories Ltd.
Kamikodanaka 1015
Nakahara, Kawasaki, 211, JAPAN

* Fujitsu Digital Technology Ltd.
Shin-Yokohama 2-3-9
Kohoku, Yokohama, 222, JAPAN.

1 Introduction

Most distributed systems consist of processes distributed amongst several different computer nodes, where each process manages its own data in a unique address space. Typically, such systems accomplish interprocess communication (IPC) by using the message-passing or remote procedure call paradigms over networks running complex protocols. Because these conventional network interfaces require significant overhead, they are not well suited to fast communication in distributed systems geared for high-performance computing.

Distributed shared memory (DSM) is an alternative IPC paradigm in which a virtual address space spanning a group of geographically-dispersed computer nodes is shared among distributed processes. In a communication system based on DSM, instead of using protocols to send and receive data, a process simply references the memory address of the data. As in multiprocessor systems, which use cache coherence and memory management mechanisms to acquire data and keep it available, the underlying DSM support architecture moves data across the network as necessary. From the perspective of the user-level process, the DSM paradigm affords several advantages over the more traditional IPC mechanisms, including better support for complex data structures and a well-understood programming model.

We have developed an architecture for implementing DSM called the Networked Virtual Memory System (NET-VMS). By achieving sensible compromises on various architectural issues and by adhering to a design philosophy of simplifying and off-loading to hardware basic DSM tasks, our communication system achieves significant improvement in performance relative to systems using traditional IPC mechanisms. Our architecture supports flexible degrees of data consistency, allowing us to examine the issue of high latency. The goal of our design is that implementations based on it should be fast and should be easily implementable in hardware.

2 Background and Related Work

Researchers have been active in exploring the area of distributed shared memory for more than a decade. Work in DSM owes much of its foundations to research into shared memory and multiprocessor machines, where such issues as cache coherence and memory management have long been looked at. But many issues in DSM research are different from those pertaining to conventional shared memory.

Conventional shared memory is intended for use in a single machine: tasks are performed in a tightly-coupled environment of multiple processors. Contrarily, tasks distributed among computers across a network are generally thought of as loosely-coupled at best. Conventional computer and network architectures promote a logical separateness between machines that are physically separated. This difference in modeling affects the perspective of the user, the programmer, and the operating system alike.

In many ways, DSM can provide the illusion of a tightly-coupled space above the network, binding together a cluster of machines into one large network multicomputer. But several drawbacks work against the illusion. Primarily, high latency caused by physical distance and communication overhead can significantly delay access to data resident in remote nodes compared to access to data that is locally resident. Synchronization also becomes more of a problem: because the DSM system is not just used to deliver shared memory but is also the system for communicating between distant machines, it must be scalable on a large order. With care and fine-tuning, bus-like techniques for cache coherence and memory management that are conventionally used in shared memory systems can usefully scale in DSM systems, as we argue below, but they nonetheless face an inevitable performance barrier.

Efficient network multicomputing (or concurrent computing, as it is also known) is a goal of potentially significant practical value. Existing workstations can remain geographically dispersed but can be clustered together to form a powerful computing machine. It should be stressed that the distinguishing characteristic of a network multicomputer compared to a conventional supercomputer is the fact that the mechanism that delivers the tightly-coupled space is also a communication system.

A valuable survey of DSM issues and algorithms is presented in [1]. Another useful comparison is presented in [2], which especially considers the issue of IPC in detail. [3] expands on this discussion of IPC and argues for treating memory as a network abstraction.

There have been several proposals to grapple with both the latency and synchronization problems. One system, the DASH system developed at Stanford [4,5], uses a directory-based caching scheme instead of broadcasting in order to improve scalability. The Mether system [6] employs a variety of techniques to reduce latency. Perhaps the most interesting techniques involve relaxing or redefining data consistency requirements. The Midway system at CMU [7] and the Munin system at Rice [8] are examples of these techniques, which are discussed generally in [9].

A general discussion of shared dataspace, relevant to tightly- and loosely-coupled modeling in distributed systems, is presented in [10]. The developers of Munin are looking at the issue of network multicomputing in a shared dataspace where data is shipped from computer to computer using a software-level DSM, as compared to function shipping using message passing and RPC techniques [11]. The same group at Rice is also working on another DSM system called TreadMarks to examine network multicomputing on standard workstations and operating systems [12]. This effort is similar in purpose to our work with NET-VMS, though our system is implemented at the hardware level.

Several software packages have been developed to deliver multicomputing environments and distributed shared memory to the user level in clusters of networked workstations. As now implemented, these packages use conventional networking interface technologies (client-server, message passing, TCP/IP, Ethernet) coupled with interface libraries to create their environments. PVM is one such popular system that was

developed at Oak Ridge National Laboratory [13]. A comparison of several of the available systems is available in [14]. We propose to use NET-VMS instead of the conventional networking technologies to deliver the distributed shared memory dataspace to the workstation clusters.

3 Architecture of the Networked Virtual Memory System (NET-VMS)

In this section, we discuss the basic architecture of NET-VMS, focusing mainly on its hardware aspects [15]. We return to the discussion of NET-VMS's architecture in our discussion of PUMA-II below.

NET-VMS is an unstructured, paged DSM that implements floating (ownership free) pages on a broadcast network. NET-VMS achieves its high performance by off-loading basic DSM management to a set of network operations implemented in a hardware controller. The controller supports strict consistency based on the write-invalidate protocol. NET-VMS supports a flexible interface between the processor and the controller that uses access type declarations to extend the programmability and manageability of the shared virtual memory.

3.1 System Overview

Figure 3.1 depicts the general design of NET-VMS. A NET-VMS distributed system consists of a group of nodes whose local memories are connected through a broadcast network by NET-VMS controllers. For the network, we assume a high-speed broadcast local area network (LAN), such as FDDI and its faster successors. Though slower LAN's like Ethernet and tightly-coupled buses are also usable, they degrade the performance and scalability of NET-VMS. In some previous implementations of NET-VMS, we have used FDDI. In PUMA-II we use a fiber optic ring running a simple protocol of our own design (discussed below).

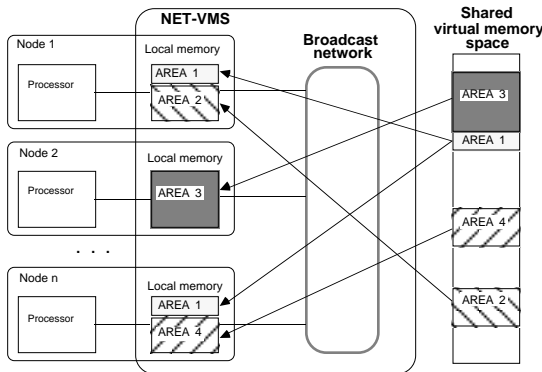


Figure 3.1 NET-VMS architecture

A node consists of a processor (or processors), a local memory, and a NET-VMS controller implemented in hardware (see Figure 3.2). The controller consists of a page manager, a shared virtual memory page table, and interfaces to the processor and the network. The page manager supervises the network and controls primitive memory consistency using the page table. The page manager maintains page consistency by changing entries in the page table according to the virtual address of a page and certain control commands contained in network frames (discussed below).

Each processor communicates with its controller through an interface. This interface consists of a bi-directional FIFO queue group at the controller (from the processor to the page manager and vice versa) and some control signals, such as an interrupt

signal to the processor and a reset signal to the controller. With this simple interface, the processor can efficiently pass instructions and receive reports asynchronously to and from the NET-VMS controller. The network interface carries out all network functions, such as the sending and receiving of network frames and the management of the network.

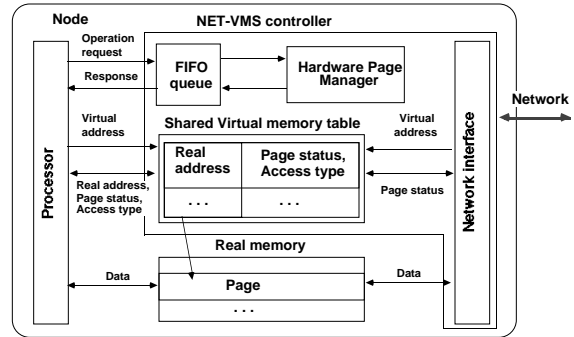


Figure 3.2 Node configuration

3.2 Network and Communication Scheme

Figure 3.3 shows the NET-VMS communication frame. The network frame consists of two major parts, a request part and a response part. The request part consists of several bytes of data and contains a virtual page address, a NET-VMS command code, and other network-related information. The response part contains a response code to the command and page data when it is needed. (Some commands do not require data movement.) The response code field is divided into sub-fields, which are used for command acknowledgment and rejection. To allow on-the-fly processing of the communication frame, there is a gap (a time lag in the frame) between the request part and the response part.

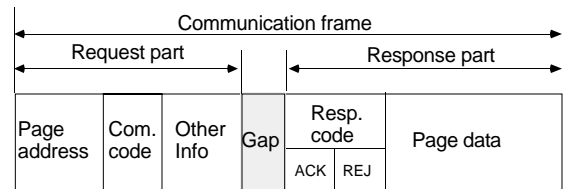


Figure 3.3 NET-VMS communication frame

To support strict consistency, NET-VMS can perform the write-invalidate protocol using two network operations: COPY and UNIFY. The COPY operation copies a shared memory page from one of the other nodes to the requesting node, while the UNIFY operation invalidates all copies in the other nodes. The functionality of the communication mechanism is flexible, and other network operations, such as UPDATE (provide all nodes with a recent copy) and BACKUP (store a copy at a specific node), are also supported architecturally.

Network operations are accomplished by serially broadcasting NET-VMS frames to all nodes. In a network operation (Figure 3.4), the processor of the requesting (local) node passes a command and virtual address to the NET-VMS controller through the FIFO interface. The controller determines the status of the page, constructs an appropriate network frame, and feeds the frame onto the network. The command passes onto the next node in the ring for processing. A responding (remote) node has adequate time during the request and gap parts of the network frame to evaluate the command, accept or reject it, and prepare for whatever data transfer is necessary. When the response part arrives at the remote node, the node modifies the response code as necessary. If a memory operation is called for, the remote node

reads data in from the frame or writes new data out to it. Any remote node capable of responding will respond to the request. One node may write over data written in earlier by another responding node. But because the data is guaranteed to be consistent, this does not pose a problem. Because processing is on-the-fly, there is no performance problem.

When the network frame returns to the local node, the response code is evaluated. If indicated, any new data is read in by the local controller into the local memory and the page table entry for the page is modified to indicate whether or not the page copy is valid and one of several copies. At the completion of a network operation, whether a success or a failure, the controller reports the command, virtual page address, and result to the processor through the FIFO interface.

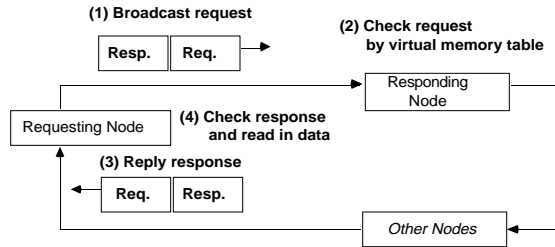


Figure 3.4 NET-VMS communication scheme

Circulating one frame around the ring completes one primitive shared memory operation (requesting and responding). The ability of the NET-VMS controller to respond in real time to a network operation without involving the remote node's processor is a distinctive feature of the NET-VMS design. With this scheme, if hardware components are sufficiently fast, it is possible to realize a communication rate comparable to the physical transmission speed of the network. Furthermore, the frame processing by the controller is independent of the processor. Thus, the broadcast frame never disturbs the processor activity of each receiving node.

3.3 Memory Management

Management of the NET-VMS pages is performed by a combination of system software and page manager hardware. The page manager performs primitive memory management related to page consistency control. The system software controlling the page manager provides non-primitive operations for shared virtual memory management (such as replacement, backup, restore, etc.).

The page manager is a finite state machine that is actuated by a NET-VMS event, such as the receipt of an instruction from the processor or the receipt of a frame from the network. The page table has an entry for each shared page. Each entry contains the following information:

- a shared memory page address;
- a physical page address pointing to local distributed memory;
- a page status indicator;
- a page access type declared by higher-level software; and
- other information necessary for page management.

NET-VMS basically uses three tags—VALID, COPY, and LOCK—to represent page status. The page manager keeps the NET-VMS shared memory consistent by referring to these tags and modifying them properly (Figure 3.5).

The VALID and COPY tags are used by the write-invalidate protocol to record the state of a page as UNIFY and COPY operations are performed. The LOCK tag is used when a page should be accessed exclusively by its own node. While the LOCK tag is asserted, the page manager rejects any network operations on the page from other nodes: the local processor of the node has

exclusive access. This feature is necessary to avoid thrashing between multiple writers and to support a structured memory.

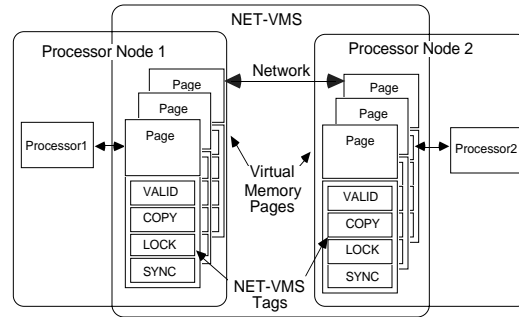


Figure 3.5 NET-VMS tag organization

Access types are used to determine how an application should access a shared page. An application declares the access type of a shared page before access using a system call. The access type declaration instructs the system software on how the page should be managed; it may be changed dynamically. For example, an application might declare an access type of "shared, read only" for an area of shared data. In the case that the application needs to change the data, it re-declares the access type for the area as "unshared, exclusive write." After finishing, the application again declares the access type, now back to "shared, read only." The area again becomes accessible to other applications.

We use access types to allow weaker levels of data consistency. Access types provide the programmer with control over how strongly consistent data must be. We discuss access types by presenting a typical example of a NET-VMS operation. In this example, we assume that an application is seeking to read a shared virtual page whose address is already known but a copy of which is not present on the node. Figure 3.6 illustrates the process explained below.

- 1) *Allocation*: The application requests the system software to allocate a physical memory area for the page with read-only status using two system calls, `valloc()` and `declare()`.
- 2) *Access*: The application attempts to access the page. Because there is no data, the application is suspended, and the page fault handler starts to run.
- 3) *Copying (software)*: The system software obtains the faulted page address and the access type and instructs the NET-VMS controller to copy the page from the network.
- 4) *Copying (hardware)*: The page manager sends a COPY frame through the network and processes the result.
- 5) *Resumed access*: Upon receiving the page manager interrupt, the system software checks the result and allows the application to resume its suspended instruction.
- 6) *Release*: After the access, the application releases the page using a `release()`.

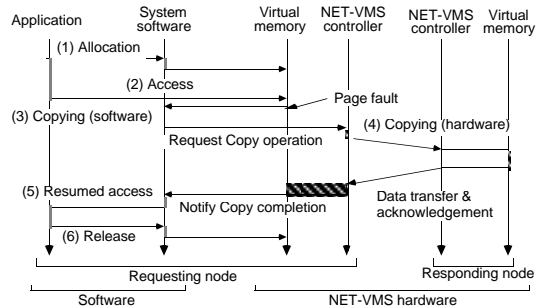


Figure 3.6 NET-VMS access scheme

4 Previous Prototypes

We have implemented five different NET-VMS prototypes, including our current prototype, which we discuss in more detail in the subsequent section. In this section, we briefly overview our previous systems and describe some results from experiments conducted on them.

Prototype-I was the first implementation of NET-VMS [16], [17]. It was a construction of simple NET-VMS controllers connected by a 100 Mbits/sec optical fiber single token ring. The prototype showed a minimum transmission delay of 330 microseconds and a maximum communication speed of 99 Mbits/sec (11.8 Mbytes/sec) for a 4 Kbyte page transmission. We attached a processor (Motorola 68010 running on 12.5 MHz) to Prototype-I and made Prototype-II [18]. Thus the NET-VMS specification of Prototype-II is essentially identical to that of Prototype-I. On Prototype-II, we implemented a simple operating system we named DAX-I and executed a parallel merge-sort program using access types to examine the impact of NET-VMS availability on distributed processing. We found that distributed scheduling based on NET-VMS floating pages achieved very fair process distribution.

The third prototype we called PUMA, but its design is dissimilar to the current prototype. PUMA encapsulated the NET-VMS frame of Prototype-II in an FDDI standard frame. The FDDI encapsulator was designed on a standard gate array. This strategy would seem to have some fundamental defects. First, the frame length becomes longer because of the encapsulation. Second, the frame passing delay in a node also becomes larger. However, our experience showed that these defects in fact had an almost negligible impact on performance.

The prototype previous to PUMA-II was an experimental shared-memory-based communication system. This prototype was designed to work with Fujitsu's SURE System 2000 [19]. SURE is a multiprocessor system based on hardware-supported message passing. We used NET-VMS to create an inter-SURE message communication system called SURE-SX. SURE-SX exhibited an at-least 2700 messages per second performance and a 1.8 millisecond average communication delay on an FDDI network using a 1.9 VAX-MIPS control processor. When 4 Kbyte messages were used, the system achieved a throughput of 88.5 Mbits/sec. Note that message-by-message acknowledging is done using NET-VMS. Comparing this result with the result using XTP (eXpress Transfer Protocol), which claims 83 Mbits/sec on an FDDI as a simulation result [20], SURE-SX demonstrates superior performance. The bottleneck on SURE-SX's performance in general was the control processor's software overhead. We estimated that by speeding up the control processor, we could achieve a proportional speed-up in message handling. We estimated that using a state-of-the-art microprocessor, a performance of over 10,000 messages per second without changing the NET-VMS hardware was possible.

5 Current Prototype: PUMA-II

The current implementation of the NET-VMS architecture is called PUMA-II. PUMA-II is distinguished from previous implementations of our architecture in that PUMA-II makes the virtual memory space delivered by NET-VMS available to a group of geographically dispersed, off-the-shelf high-performance workstations running a variant of the UNIX operating system. The PUMA-II specification separates the NET-VMS system components from the workstation processor and operating system through a convenient interface abstraction, making extension and modification easy. We used a similar approach at the interface between the NET-VMS controller and the network substrate, thus making feasible eventual migration to other networking platforms. In PUMA-II, we have adhered to our philosophy of off-loading networking tasks to low-level hardware.

5.1 PUMA-II Implementation

The PUMA-II system can consist of up to 255 nodes. Each node is a SPARCStation 2 (SS2) workstation equipped with a VMEbus and VME Subsystem Bus (VSB) expansion. Connected to the SS2 through the VMEbus are the NET-VMS memory and network controller boards. Within each node, the NET-VMS boards are also directly connected to each other through the VSB. The network controller boards are connected to each other in a ring-topology optical-fiber broadcast network operating at 100 Mbits/sec. The virtual address space of the system is 1 Gbytes, and each node may have up to 256 Mbytes of real memory. Pages are 4 Kbytes in size.

The network controller board is a printed circuit board of our own design mounted with large, high-speed programmable logic devices (PLDs) and other components. We used the PLDs to implement a set of highly specialized and interconnected finite state machines. This modular state-machine design allows easy implementation and upgradability of functionality and efficient, parallel operation of basic page-processing tasks.

The SS2 workstation runs the SunOS, a variant of UNIX. The interface between the operating system and the PUMA-II hardware components is managed by a device driver. The device driver controls the high-level operations of the hardware components through a command and response protocol. The device driver also manages memory functions, and is capable of either mapping pages directly from the PUMA-II memory board or copying pages into main memory. A library of system calls interface user processes with the device driver.

For our test platform, we have developed a simple device driver that allows us to evaluate the basic behavior of our hardware system. This device driver supports the fundamental NET-VMS network operations, COPY and UNIFY, as well as the UPDATE network operation. We have designed the device driver to present NET-VMS to the operating system as a read-only filesystem—that is, as a non-writable shared RAM disk. This implementation allows us to interact with PUMA-II through familiar kernel mechanisms for filesystem manipulation while also easily evaluating our underlying hardware mechanisms.

The software-hardware interface consists of a simple but powerful command and response mechanism. In the address space of the device driver, two 32-bit locations are mapped respectively through the VMEbus to the input and output registers of the command and response FIFO's on the PUMA-II network controller board. A signaling interface is also mapped between the board and the device driver. To issue a command, the device driver writes the command to the command address and signals the device driver. The board then registers this command into the command FIFO. Similarly, to report a response, the board registers the response on the response FIFO and signals the device driver, which then copies the response from the response address. The signaling interface also includes some diagnostic mechanisms for assessing and resetting the state of the board.

The design of the shared RAM disk is modeled after a conventional RAM disk. In a conventional RAM disk, a section of main memory is reserved and is modeled to appear as a filesystem. A read from the filesystem results in a series of copies managed by the device driver from main memory to a kernel buffer to user space. A write to the RAM disk operates in a reverse fashion.

In the shared RAM disk, the device driver maintains a table of pages that are immediately available in the local PUMA-II memory. The main store node always has all pages available. Other nodes manage their local memory, which operates as a cache. If a read request is conveyed by the kernel, the device driver determines whether the page is available locally. If it is, as will always be the case for the main-store node, the device driver copies the page into the kernel buffer. If it is not, which may be the case for a caching node, the device driver issues a COPY network operation command to the PUMA-II hardware system. The PUMA-II hardware then issues a NET-VMS frame onto the network to retrieve a copy of the page from a remote node that has a copy of the page. If a successful response is reported, the device driver copies the newly-

acquired page into the kernel buffer. If necessary, an older page is discarded according to the direct mapping cache algorithm. (We chose this algorithm for its simplicity and speed and because it was sufficient for our testing purposes. A more robust implementation would use other replacement algorithms like LRU.)

Under SunOS, direct access by device drivers to kernel buffers is not allowed. Because of this, modifications by other nodes to a local page through network operations, while achievable by PUMA-II itself, could nonetheless render any data in the kernel buffer inconsistent. This is unlike a conventional RAM disk, where data in main memory can only be modified by the device driver itself, thus guaranteeing consistency to data in the kernel buffer. Write support for the shared RAM disk is not supported.

5.2 Performance Evaluation

In this section, we present two sets of results. The first is a discussion of PUMA-II basic performance. The second is an evaluation of the shared RAM disk performance. We interpret the results and relate them to our previous prototypes. Also, we elaborate on our interpretation by suggesting improvements in the PUMA-II hardware implementation.

Basic performance of PUMA-II was evaluated using the mapping functionality of the VMEbus driver provided with the SS2. This device driver offers only limited functionality and cannot support the wide range of features we aim to implement on PUMA-II. For example, the device driver uses the existing mmap() system call, which has a limited parameter set. Because of this, its usefulness is limited to access to a page by only one process at a time. Despite such limitations, we were able to use the device driver to characterize the basic performance of PUMA-II.

We developed a test program to sequentially access a section of PUMA-II memory. We were able to preset the local memory to either contain or not contain the data to be accessed. In this way we were able to induce sequential page movement from remote nodes to the local node across the PUMA-II network.

Running the test program, we examined the performance at the hardware level. For the COPY and UPDATE network operations, the transfer speed results were the same, as was to be expected. As for delay, we found that PUMA-II achieved a 340 microsecond transfer rate per 4 Kbyte page across the network, which is approximately 11.5 Mbytes/sec. Because we operate our network at a rate of 100 Mb/s, this result is an effective utilization of approximately 96 percent of the available bandwidth. The UNIFY network operation does not transfer data, only a command, and it required 3 microseconds to complete.

At the software level, our test program sequentially processed 40 Mbytes of memory accessed in 32-bit word (4 byte) increments (Figure 5.1). PUMA-II local access involves the combined overhead of reading from the local PUMA-II memory and the VMEbus transfer, while PUMA-II remote access involves the additional overhead of the network transfer of paged data (each page was retrieved separately). For comparison, sequentially processing 40 Mbytes of data retrieved from a direct mapping to main memory required 26.0 seconds. When the 40 Mbytes of data was resident in local PUMA-II memory, the time required was 31.5 seconds, 21 percent longer than the main-memory case. In the case of accessing remote memory, the time required was 35.0 seconds, 34 percent longer than the main-memory case.

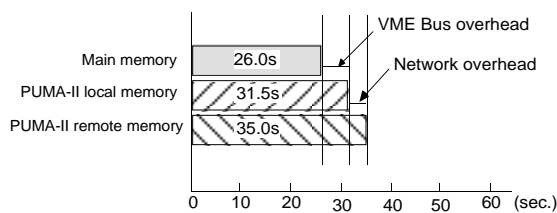


Figure 5.1 PUMA-II Software Performance

To evaluate the performance of the shared RAM disk, we used another simple program to sequentially read through data on the filesystem. Similar to evaluating the basic PUMA-II performance, we were able to preset the local PUMA-II memory to contain or not contain the file data. When the local PUMA-II memory was set to contain the file data and those pages were registered in the device driver's page table, the pages were copied by the device driver through the VMEbus and a kernel buffer to the test program, using the standard UNIX filesystem read() system call. When the file data was preset as not resident, the device driver first instructed the PUMA-II hardware to retrieve the pages from remote nodes before executing the copy into user space.

For comparison, we created a conventional RAM disk that used main memory. We found that the conventional RAM disk achieved a read rate of 320 ms per megabyte. In the case of the shared RAM disk reading from local memory, overhead was contributed by additional device driver complexity, local memory access, and VMEbus transfer. For this case, the read rate was 520 ms per megabyte. In the case of the shared RAM disk reading from remote memory, with additional overhead contributed by retrieving each page across the network, the read rate was 610 ms per megabyte. Thus, for the shared RAM disk using the network for all accesses, in comparison with the conventional RAM disk using main memory, PUMA-II network operations accounted for 15 percent of the overhead while device driver, local PUMA-II memory reading, and VMEbus transfer accounted for 33 percent of the overhead.

To demonstrate the value of the shared RAM disk concept, we also looked at the read rates of a SCSI hard disk (single-user, single-process access) and of NFS (running on a lightly-loaded Ethernet subnet). The transfer rate in the SCSI hard disk case was 880 ms per megabyte, and in the NFS case was 1230 ms per megabyte.

The basic performance of PUMA-II is impressive. For remote memory access, in comparison with main memory access, PUMA-II network operations accounted for only 10 percent of the test program processing time while reading from local PUMA-II memory and VMEbus transfer accounted for 16 percent of the processing time. Together, the overhead added by PUMA-II was 26 percent. The actual network operations made up 39 percent of the overhead, while the remainder of the overhead (61 percent) was contributed by the slow memory board and VMEbus. Below we discuss methods we estimate will reduce the overhead by a factor of three.

These results hold up in the case of an actual UNIX-level application, the shared RAM disk, even though we mapped the data directly and relied on copying. Our test program had very little processing overhead, simply reading the data from the disk (i.e. the DSM). Access to file data resident on remote nodes through the shared RAM disk took nearly twice as long as access through a conventional RAM disk, where the file data was stored in main memory. The network operations took up 31 percent of overhead, with the remaining 69 percent being contributed by local PUMA-II memory reading, VMEbus transfer, and device driver complexity. While we have only preliminarily tested the shared RAM disk, our results demonstrate that PUMA-II can be used efficiently at the user-process level as a communication system, in comparison with other methods.

5.3 Proposed Improvements

The main bottlenecks we have identified in the hardware are the slowness of memory access from the local PUMA-II memory board through the VMEbus and the slowness of the network interconnecting the PUMA-II controller boards. We believe that we can migrate to an Sbus-based interface between the processor and the controller board, as well as utilize a local PUMA-II memory board with faster memory access speeds. The memory access rate through the VMEbus is 7.3 Mbytes/sec. By using a 20 Mbytes/sec Sbus and a memory board with a 25 MHz (4-byte, 5-cycle) access rate that is faster than the Sbus, we estimate we can transfer at the Sbus rate, and thus increase our access rate by a factor of 2.75.

Regarding the network, we believe we can improve our network transmission by a factor of 5.2 by moving to a 1 Gbits/sec Fibre Channel network in a 16-node, 1.6 km ring configuration. Coupled with channel degradation caused by the 1.6 km distance, we believe we can attain an effective bandwidth of 506 Mbits/sec.

We have considerable experience with the hardware element of NET-VMS from previous prototypes, and our results have confirmed that we can deliver the NET-VMS environment with efficiency to the UNIX domain. Our simple device driver has let us explore some of the uses of NET-VMS, though it has many deficiencies.

6. Network Multicomputing

In our work, we have demonstrated the general performance of the NET-VMS architecture as a communication system. We have not yet determined how NET-VMS performs under loaded conditions in a complex computing environment, where several distributed processes are competing to read data that requires access through the network. This is a problem faced by all DSM researchers, and more generally, also by distributed processing researchers. Namely, the question is how to provide adequate communication resources that can scale to a useful degree while also achieving maximal efficiency.

To examine these issues, we are now working on coupling our DSM system with the PVM software package. PVM was originally designed to use message-passing IPC between processes and to use a traditional network interface for inter-nodal communication. Running on top of Ethernet and using message passing, PVM achieves about a 0.5 Mbits/sec data transfer rate [14]. PVM was later extended to allow the use of intra-nodal shared-memory IPC. We are now porting this version of PVM to our system.

The address space of our DSM can be mapped directly into user space, and the shared-memory version of PVM can operate directly on this mapped space. The shared-memory version of PVM differentiates between intra-nodal IPC, which it accomplishes through sharing pointers among processes, and inter-nodal IPC, which it accomplishes through its message passing interface. We estimate the port will mainly involve modification of how PVM differentiates between local and remote memory.

Considerable work is being done currently in evaluating the performance of the available software packages being used for network multicomputing. We estimate that with PVM we will be able to achieve a throughput of 5 Mbytes/sec using a PUMA-II 100 Mbits/sec network.

7 Conclusion

The PUMA-II prototype is the first of our implementations to deliver NET-VMS directly to the UNIX-domain user level. Our preliminary results show that PUMA-II retains the efficient characteristics of our previous prototypes while opening up an entire realm of questions to explore at the systemic level. Significantly, PUMA-II delivers to the user level a tightly-coupled space that is geographically distributed across several processing nodes. Exploring IPC efficiency in this domain using bench marking software with the PVM system should be revealing about the practicality of DSM systems in distributed processing.

References

[1] B. Nitzberg and V. Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms," IEEE Computer, vol. 24, no. 8, pp. 52-60, August, 1991.

[2] M.-C. Tam, J.M. Smith, and D. J. Farber, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems," ACM Operating Systems Review, vol. 24, no. 3, pp. 40-67, July, 1990.

[3] G.S. Delp, D.J. Farber, R.G. Minnich, J.M. Smith, M.-C. Tam,

"Memory as a Network Abstraction," IEEE Network, pp. 34-41, July, 1991.

[4] D. Lenoski, et al. "The Stanford DASH Multiprocessor," IEEE Computer, vol. 25, no. 3, pp. 63-79, March, 1992.

[5] D. Lenoski, et al. "The DASH Prototype: Logic Overhead and Performance," IEEE Transactions on parallel and Distributed Systems, vol. 4, no. 1, pp. 41-61, January 1993.

[6] R.G. Minnich and D.J. Farber. "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory," In Proceedings of the 10th IEEE Distributed Computing Systems Conference, pp. 468-475, 1990.

[7] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. "The Midway Distributed Shared Memory System," In Proceedings of the '93 CompCon Conference, pp. 528-537, February, 1993.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. "Implementation and Performance of Munin," In Proceedings of the 13th ACM Symposium on Operating Systems Principles, pp. 152-164, October, 1991.

[9] P.W. Hutto and M. Ahamad. "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," In Proceedings of the 10th IEEE Distributed Computing Systems Conference, pp. 302-309, 1990.

[10] G.-C. Roman and H.C. Cunningham. "A Shared Dataspace Model of Concurrency: Language and Programming Implications," In Proceedings of the 10th IEEE Distributed Computing Systems Conference, pp. 270-279, 1989.

[11] J.B. Carter, A.L. Cox, S. Dwarkadas, et al. "Network Multicomputing Using Recoverable Distributed Shared Memory," In Proceedings of the '93 CompCon Conference, pp. 519-527, February, 1993.

[12] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," In Proceedings of the '94 Winter USENIX, pp. 115-131, January, 1994.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and Sunderam, V. "PVM 3 User's Guide and Reference Manual," Report ORNL/TM-12187, Oak Ridge National Laboratory, May, 1993.

[14] C.G. Douglas, T.G. Mattson, and M.H. Schultz. "Parallel Programming Systems for Workstation Clusters," Report YALEU/DCS/TR-975, Yale University, Department of Computer Science, August, 1993.

[15] A. Jinzaki. "A Fast Distributed Shared Virtual Memory System: NET-VMS," Fujitsu Scientific and Technical Journal, vol. 29, no. 3, pp. 286-295 (1993).

[16] A. Jinzaki and R. Yatsuboshi. "NET-VMS: A Computer Network Architecture for Distributed Computer Systems," IEEE Tokyo section "Denshi Tokyo", vol. 28, pp. 14-17 (1989).

[17] A. Jinzaki and R. Yatsuboshi. "A Distributed Single Level Virtual Memory using a Broadcasting Network (in Japanese)," Papers IECE Japan, CPSY86-20, pp. 1-11 (1986).

[18] A. Jinzaki and M. Higuchi. "Declarative Access Control Scheme on Networked Virtual Memory System (in Japanese)," Papers IPS Japan, vol. 30, no. 20, pp. 1612-1619 (1989).

[19] A. Kabemoto and H. Yoshida. "The Architecture of the Sure System 2000 Communication Processor," IEEE Micro, vol. 11, no. 4, pp. 28-31, pp. 73-78 (1991).

[20] D. Tsao. "FDDI: Chapter Two," Data Communications, vol. 20, no. 13, Sept. 21, 1991, pp. 59-70 (1991).