THE VIDEO-RAM MULTIPROCESSOR ARCHITECTURE

R.D. Hersch, F. Maddaleno, C. Nicks, M. Burki

Swiss Federal Institute of Technology, Lausanne

Loosely coupled multiprocessor systems communicate by message passsing. Traditional implementations of message passing architectures are based on communication controllers accessing local memory through DMA channels. This paper describes a completely new communication mechanism based on the transfer of messages through dual-port memories. The proposed architecture takes advantage of the dual-port nature of dynamic video-ram ICs in order to ensure efficient and deadlock-free transfer of messages between communicating processes.

Keywords: multiprocessor, communication memory, message-passing hardware

1. Introduction

Modern cost effective multiprocessors distinguish themselves from more traditional designs by making extensive use of powerful microprocessors [1]. One of the most important issues concerns the interconnection topology for efficient communication between tens, hundreds or thousands of microprocessors. A small number of processing units can be interconnected using industrial parallel buses like the VME bus [2]. More advanced very high throughput parallel buses can connect up to 30 processing units [3]. High cost and throughput bottlenecks of parallel buses led to the design of cube architectures. Conventional cube architectures are characterized by a more than linear increase of the number of communication channels relatively to the number of processing nodes. They allow graph- or tree-oriented problems to be mapped into the cube processors.

Advances in VLSI logic led to the integration of communication channels within the central processing unit [4], [5]. These integrated circuits also provide the basis for very compact multiprocessor systems designs. However, these systems lack object code compatibility with common microprocessors, like the 68´000 or the 8086 processor families. Therefore, applications running on mono-processor workstations cannot be easily moved to these new multiprocessor systems.

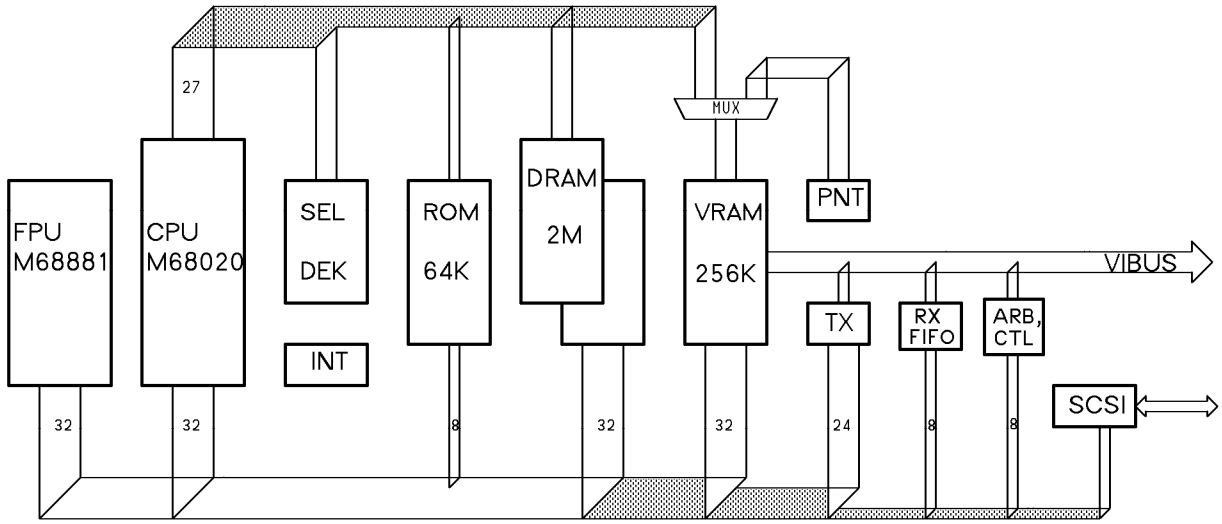
The authors of this paper have taken an innovative approach by designing a new communication mechanism, which consists of a communication bus linking together dual-port memories, each belonging to another processing unit. This approach, materialized by the use of serial input/output ports of Video-Rams, leads to the new concept of *smart communication memories*.

A first implementation required 35 discrete circuits for the communication logic only. It is expected, however, that advances in VLSI should allow the integration of complete message communication and storage functionality into one circuit.

R.D. Hersch, F. Maddaleno, C. Nicks, M. Bürki, "The Video-Ram Multiprocessor Architecture", Proceedings Euromicro 88, Zürich, (Winter, Schumny, Eds.), Elsevier Science Publ., 1988, 503-510

2. Video-Ram Multiprocessor Architecture

One processing pool consists of a set of processing units (sites), each having a conventional 32 bits CPU, a convenient amount of memory and input-output capabilities (figure 1). In addition to these elements, each processing unit also contains a bank of Video-Rams and associated communication logic.



vram1

Fig. 1 Architecture of one processing unit

The dual-port nature of Video-Rams [6] leads to a very convenient implementation of an independant communication channel responsible for the transfer of messages between processing units. Viewed from the CPU, a Video-Ram behaves, from its parallel access port, exactly like dynamic RAM. A message which is ready to be sent is stored by the local processor in one or more successive Kbyte blocks.

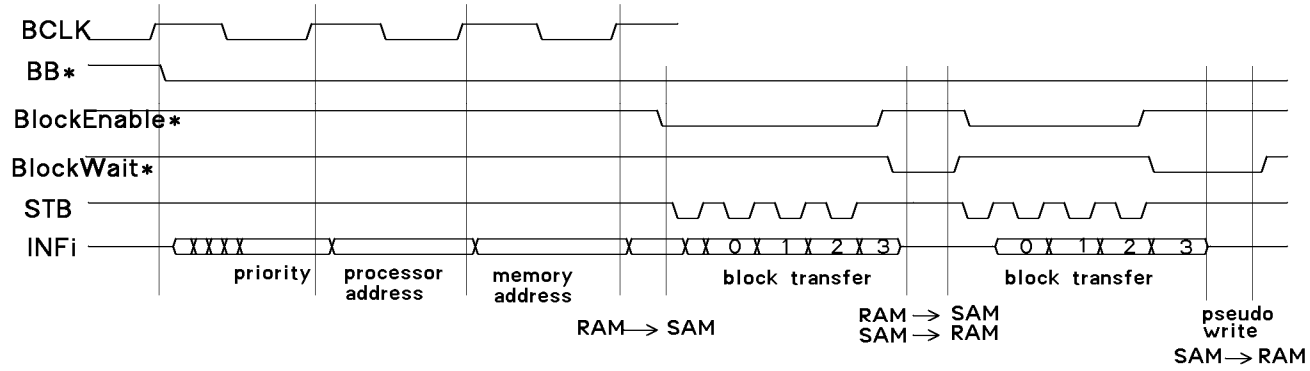
The communication logic associated with the serial Video-Ram port is responsible for arbitrating access to the common bus (Vibus), for generating message header information, for serializing the message data stored in Video-Ram and for transfering the message to the desired destination location.

A message transfer is initialized by writing into the communication logic the type of transfer, the destination site, and the destination location corresponding to the receiving process. After parameter initialization by the local processor, the transmission logic automatically arbitrates its own access to Vibus. Once the latter has been granted, the communication logic transmits the message header, reads the data from the serial Video-Ram port, serializes it and transmits it over the 8 bit wide Vibus. The message, together with the header specifiying destination site, location and transfer type is stored at the right location by the communication logic of the receiving site.

On the emitting side, an interrupt tells the processor that the message has been sent and that the corresponding buffer place has become free. On the receiving side, an interrupt signals the receipt of a new message. Associated header information specifying destination process and message type has been written in a FIFO memory. Therefore, incoming messages are served asynchronously by the receiving processor.

Transmission of messages takes place over Vibus. Vibus is a reduced semi-synchronous byte wide bus for fast transfer of data blocks (4 to 1024 bytes, 20 Mbytes/s). One Vibus transaction consists of an arbitration cycle, an addressing cycle, and several block transfer cycles. Arbitration is obtained via a self-selection circuit. At arbitration time, the 8 data lines

are used as priority lines. In the control phase following arbitration, the winning emitter places destination site and destination process identifiers on the bus. Finally, variable length data blocks are transmitted. Handshake signals *BlockEnable* and *BlockWait* synchronize the exchange of data blocks between devices. Within each data block, the transfer of bytes is done synchronously, using a strobe signal STB (figure 2).



vram2

Fig. 2 Message transfer through Vibus

3. Communication control logic

The communication control logic must transfer the information from the Video-Ram to Vibus and vice versa, by receiving commands from the processor and starting Video-Ram access cycles. Before describing the control logic let us briefly introduce the concept of Video-Rams.

A Video-Ram is a dynamic dual-port memory with a parallel port allowing random read and write accesses (RAM), and a serial port containing a shift register from which a whole row of memory can be sequentially read or written (SAM, serial access memory).

The serial access memory can be loaded in parallel with the contents of a RAM row, or it can be copied into a RAM row. This shift register can also be serially read or written from the serial port. The copy operation from RAM to SAM is named Read Transfer. It puts the shift register into output mode. The transfer from SAM into a RAM row is named Write Transfer and puts the shift register into input mode. A Pseudo-write operation allows the shift register to be placed into input mode without destroying the contents of any memory row. The Read, Write and Pseudo-write operations are controlled by special Video-Ram cycles carried out on its parallel port.

The Video-Ram circuit used in this implementation are organized in 64 kbytes by 4 bits (256 memory rows). The SAM is 256 words long and 4 bits wide. The SAM is accessed through the serial port in accordance to a serial clock and a serial enable signal.

The hardware which handles communications over Vibus can be decomposed into the following functional blocks (Figure 3):

- Video-Rams
- processor interface
- Vibus arbiter
- transmitter controller
- receiver controller
- Video-Ram serial port controller
- Video-Ram parallel port controller

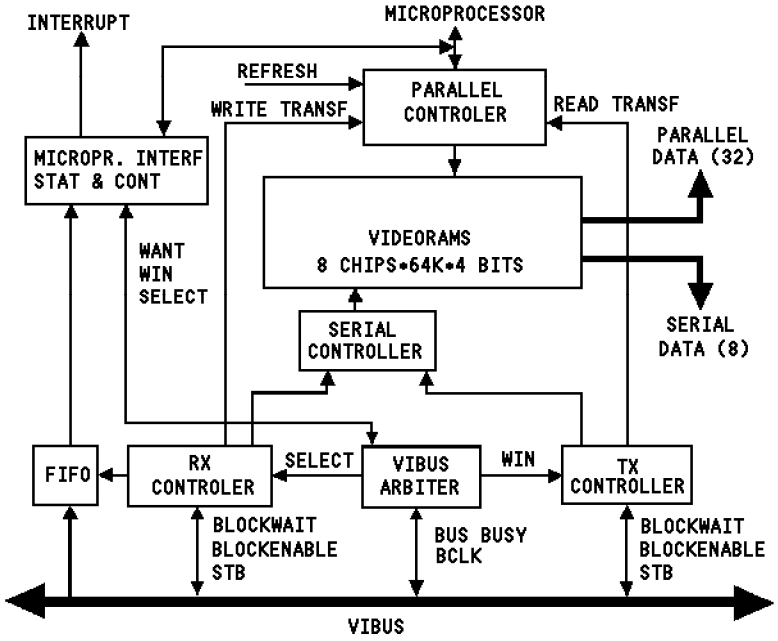


Fig. 3 Block diagram of the control logic

Eight 64k * 4 bits Video-Rams enable us to build a 32 bit wide memory (processor side). On the serial side, the SAM registers are connected to obtain an 8 bit wide data path. The processor interface contains the control and status registers and the interrupt logic. In order to prepare a transmission request, the processor first programs the message source address and its length. Then the processor writes into a second register the identifier of the destination processor and the message destination address. To start the message transmission, the processor interface activates the WANT signal asking the Vibus arbiter to request the bus.

The bus arbiter is controlled by a finite state machine which rules arbitration and site addressing. The arbitration cycle is synchronous with the clock signal BCLK and lasts one clock period (200 ns). The Bus Busy signal (BB) indicates that a transaction is taking place on Vibus.

In the BCLK clock period following the arbitration, the winner places on the data lines the address of the destination processor(s), using geographical addressing. In the following BCLK period it issues the message destination address

After these operations, the arbiter issues a signal to the Transmitter Controller (TXC) indicating that the transmission can start. TXC asks the parallel port controller to copy a specified row of the RAM memory to SAM. It then activates the BlockEnable bus signal, waits for BlockWait signal inactive, and starts data transmission. In order to feed the 8 bit wide Videobus, two Video-Ram chips are read at the same time. Multiplexing is carried out by the serial controller. At the end of a row, TXC checks whether there are other rows to send. If necessary, it requests further Read Transfers and data transmissions.

On the receiving side, the arbiter also acts as the Vibus address decoder. When selected, it asks the Receiver Controller (RXC) to store the destination address. After being selected, RXC immediately requests the parallel controller to perform a Pseudo-write Transfer in order to put the Video-Rams in input mode. At that point the system is ready to accept the first row of data which enters the SAM. The routing of incoming bytes into the Video-Rams and the generation of Video-Rams´ serial control signals are provided by the serial controller. At the end of a row (indicated by BlockEnable becoming inactive), RXC requests a Write Transfer at the address indicated in the previous addressing phase. During the Pseudo-write and the Write Transfers, the receiver prevents the transmitter from sending data by activating the BlockWait signal. The receiver recognizes the end of a transfer since Bus Busy once again becomes inactive. After the data reception, RXC places in the FIFO the start address of the newly

arrived message and sends (through the processor interface) an interrupt request. The processor may fetch the message address from the FIFO.

The controller handling accesses to the parallel Video-Ram port is basically an arbiter implemented with a state machine. It arbitrates between 3 different requests (in increasing order of priority):

- processor access (default winner)
- refresh
- Read, Write or Pseudo-write Transfer

This arbiter also controls the circuitry generating RAS, CAS and other Video-Ram signals.

All the controllers are implemented with PALs, mostly working with the same 20 MHz clock. Great care has been taken to synchronize the asynchronous inputs in order to limit the probability and risks of metastable states. The current implementation requires about 35 integrated circuits. The different state machines as well the pointers and counters could easily be integrated into an application specific integrated circuit, thus dramatically reducing the number of required integrated circuits.

Vibus bus does not use any kind of pipelining: this choice reduces the number of bus lines and drivers, but it increases the total time required for consecutive (queued) transactions. This is not a serious drawback, since queued bus transactions are not frequent. Moreover, the arbitration and selection cycles require about 600 ns. Data transfer takes for each byte 50 ns. If the transferred block is longer than 16 bytes, the total transaction time is mainly given by the data transfer time.

Vibus allows broadcast cycles. This feature is obtained by using geographical addressing and suitable handshaking signals. For this purpose, BlockWait is an active low signal controlled by open collector drivers permitting the slowest receiver to delay the bus transfers.

4. Buffer allocation

In a multiprocessor multi-tasking environment, buffer allocation for synchronisation and communication is critical. Deadlock free communication between any pair of tasks should be ensured. The software concept for task intercommunication is based on the *SendAndWaitReply* primitive [7], which is itself derived from the V-Kernel interprocess communication primitives [8].

The following communication primitives are used :

SendAndWaitReply (in:DestProcess, in:DestSite, in:MessageContent, out:AnswerContent)

> The client process, executing on *SrcSite* sends a message to the server process *DestProcess* located on site *DestSite*. After having sent its message, the client process is suspended.

Receive(out:SrcProcess, out:SrcSite, out:MessageType, out:MessageContent)

> The server process is suspended until the arrival of a message meant for it. After arrival and consumption of the message, an acknowledge is sent back to the client process. The server process resumes execution.

Reply(in:SrcSite, in:SrcProcess, in:AnswerContent)

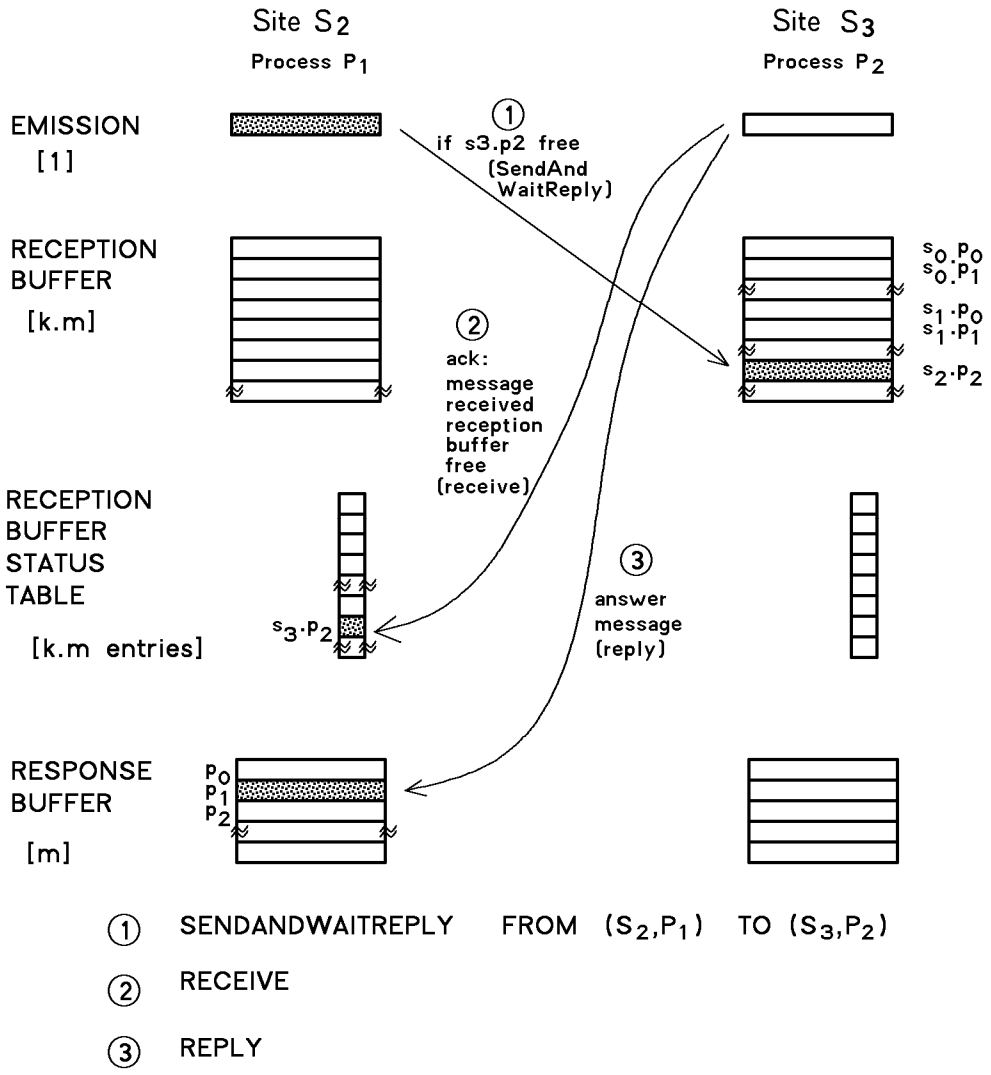
> The server process sends an answer message to its client process. This answer message allows the client process to resume execution.

Each process can be a client or a server process. Therefore, a buffer organisation is proposed, which allows any process running on any site to communicate with any other process.

For sending messages, exactly one *emission buffer* is allocated on each site. The emission buffer will only be used, if the client process is sure that the message can be received on the server´s process site. A *ReceptionBufferStatusTable* located on each site allows the local processes to maintain exact knowledge of the state of the receiving buffers on the other sites. This table contains the empty/full status for each receiving buffer of server processes able to communicate with the considered site. *Response buffers* provide the space to store answer messages for each resident client process.

Each local server process must be able to receive messages from any other process located on any other site. It is only necessary to allocate one buffer per communicating site for each server process. Communication from two client processes residing on the same site to one server process on another site will take place sequentially. *Reception buffers* allow the storage of messages from each site for each local server process.

Let us consider a multiprocessor system with k sites S_0, S_1, ...S_{k-1} each running m processes P_0, P_1, ...P_{m-1}. On each site, there will be one emission buffer, m response buffers, one ReceptionBufferStatusTable with k·m entries, and k·m reception buffers (fig. 4).



VRAMBUF

Fig. 4 Buffer allocation on sites S_2, S_3

These communication primitives are currently being tested on the target multiprocessor hardware.

5. Multi-pool systems

Only a limited number of processing units can reasonably be hooked on a common byte-wide parallel bus like Vibus. Further extensions of the number of processing units require bridges between processor pools. A bridge is a processing unit having two or more Video-Ram communication ports. Bridges are responsible for message passing between processor pools.

Many possible connection schemes could be proposed for the interconnection of processing pools. Cube architectures represent interesting interconnection structures for solving different kind of graph- and tree-oriented problems [4]. By considering one processor pool as a cube edge, it becomes possible to build a mixed architecture consisting of parallel common buses for communication on the pool level and of cube vertices for interpool message passing. A 3-dimensional cube can be constructed with 12 pools (edges) and 8 bridges (vertices) (figure 5).

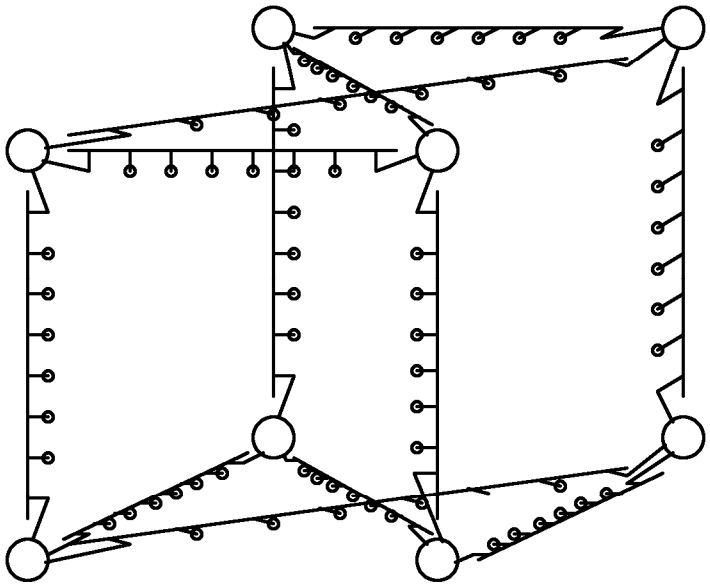
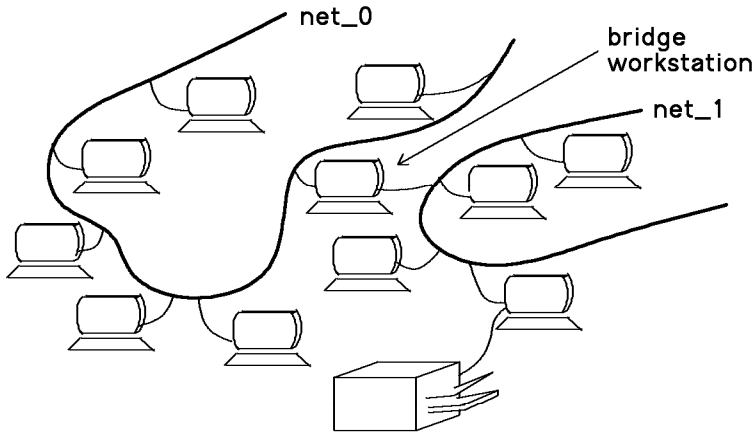


Fig. 5 Optimised three-dimensional pool-cube topology

The optimised pool-cube topology provides a framework for 72 processing units distributed in 12 pools interconnected by 8 bridge processors. A maximum of 2 intermediate bridge processors is required to pass a message from any processing unit to any other processing unit.

6. Distributed processing

The previously described multi-processor system is to be used as the basic hardware of a multi-processor workstation. Operating system and applications running in today´s distributed environment (figure 6) consisting of mono-processor workstations interconnected by local area networks [9] should also run without modifications on the currently developed multi-processor workstation (figure 7).



vram4

Fig. 6 Interconnected mono-processor workstations

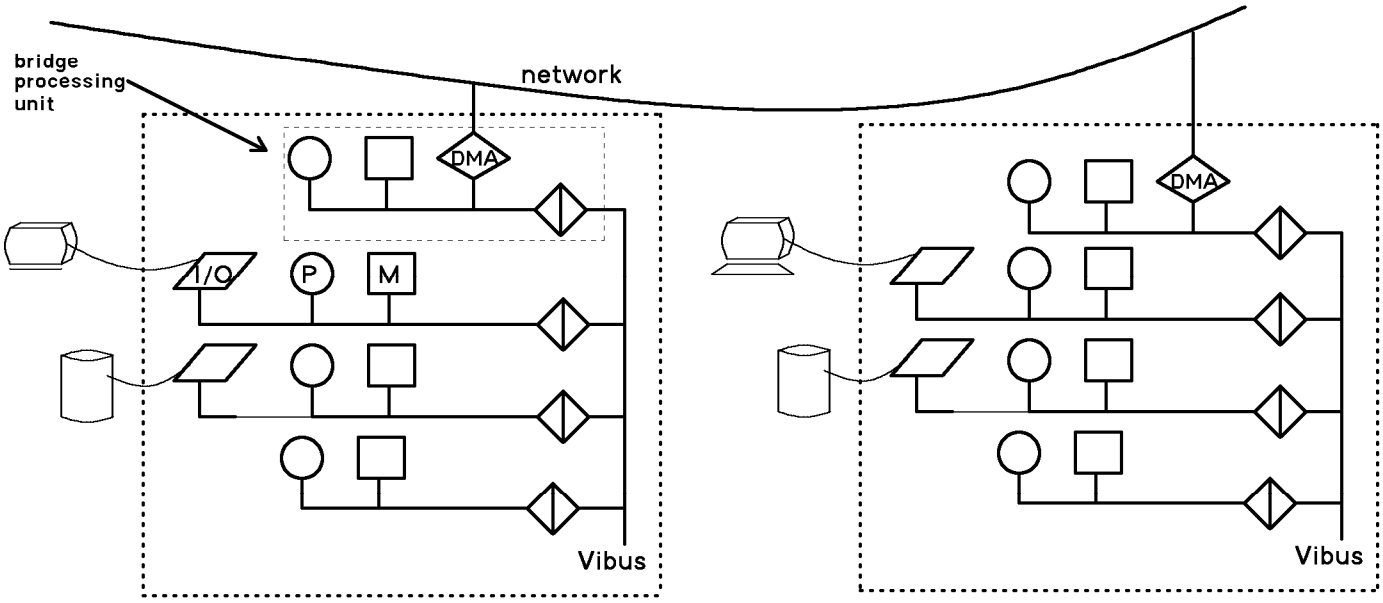


Fig. 7 Network of multiprocessor workstations

Our home made operating system [10] supports network-wide system calls by message passing and message forwarding mechanisms. The Vibus communication facility appears to the kernel like any other network. Each processing unit contains a copy of the real-time kernel, the file operating system and the required I/O drivers. Only the network driver must be changed to fit Vibus functionality. In a first approach, tasks are statically distributed by the command line interpreter running on the processing unit responsible for human input-output. A description associated with each task specifies the required resources in terms of memory space, processor type and input-output needs. At task distribution time, the system load is checked and the task is assigned to the processing unit matching the basic needs of the process and currently carrying the lightest load.

7. Conclusions

The new proposed multiprocessor message passing mechanism is based upon smart communication memories. Communication memories contain the logic to transmit stored messages or to receive messages from other locations through a common narrow high-speed bus structure. This new architectural concept releases the processor from low-level communication tasks and does not require processing power for communication purposes. Unlike most DMA communication channels, communication through smart memories does not slow the local processor down. The proposed communication scheme ensures reliable transmission of messages. Message buffer organization and communication protocols allow deadlock-free non-repetititive single message transmission [7].

Pools of processing units are interconnected by bridge-processors communicating with two or more pools. Bridge processors can be the vertices and processor pools the edges of multi-pool cube structures.

Today´s implementation of smart communication memories is based on dual-port Video-Rams. The associated communication logic requires about 35 integrated circuits.

Acknowledgments

We would like to thank Prof. Nicoud for his initial ideas concerning the usefulness of Video-Rams. We would also like to thank Prof. Schiper and Roland Simon for the specification of the software communication protocol.

References :

[1] J. Bond, "Parallel-processing concepts finally come together in real systems", Computer Design, June 1987, pp 51-74.

[2] "VME bus Specification Manual," rev C, VITA group, 1985

[3] R.A. Olsen, et al, "Messages and Multiprocessing in the ELXI System 6400," Compcon 1983, San Francisco, 1983

[4] J.P. Hayes, et al., "A Microprocessor-based Hypercube Supercomputer", *IEEE Micro,* October 1986, pp. 6-17

[5] Paul Walker, "The Transputer," Byte, May 1985, pp. 219-235

[6] J.D. Nicoud, "Structure and Applications of Videorams", *IEEE Micro*, February 1988, pp. 8-27

[7] A. Schiper, et al., "Efficient implementation of rendez-vous", Internal Report, Dept. Mathematics, EPFL, May 1988, submitted to the British Computer Journal

[8] D. Cheriton, "The V Kernel, A Software Base for Distributed Systems", IEEE Software, April 1984, pp. 19-42

[9] R. Sommer, "A real-time protocol for a sub-local network", *Local Networks and Distributed Office Systems*, Online Conference Proceedings, London, May 1981

[10] D. Dumoulin, D. Roux, "FOS," published by Epsitec-System SA, Belmont, Switzerland