MEMSY
A Modular Expandable Multiprocessor System

F. Hofmann, M. Dal Cin,
A. Grygier, H. Hessenauer, U. Hildebrand,
C.-U. Linster, T. Thiel, S. Turowski

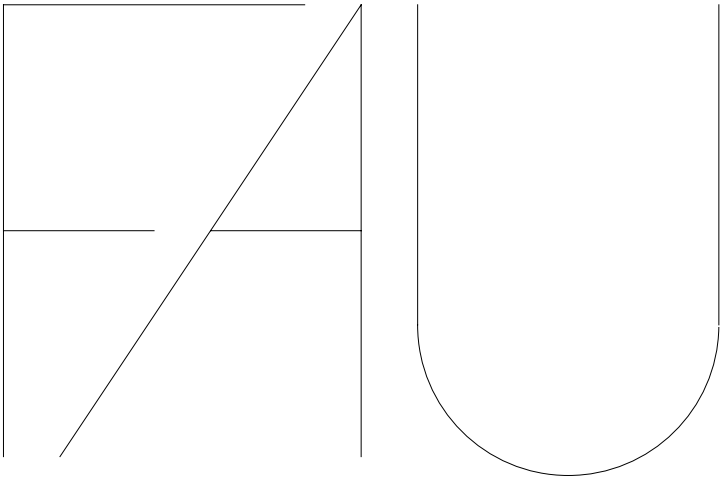
Oktober 1992 TR-I4-8-92

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper was also published as:

F. Hofmann, M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, C.-U. Linster, T. Thiel, S. Turowski: “MEMSY — A Modular Expandable Multiprocessor System”; *Parallel Comp. Architectures: Theory, Hardware, Software, and Appl.* – SFB Colloquium SFB 182 and SFB 342; A. Bode, H. Wedekind [Eds.], (Munich, Oct. 8-9, 1992); Lecture Notes in Comp. Sci.; Springer, Berlin; to appear 1993

MEMSY
A Modular Expandable Multiprocessor System

F. Hofmann
hofmann@informatik.uni-erlangen.de
M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand,
C.-U. Linster, T. Thiel, S. Turowski

University of Erlangen-Nürnberg
IMMD, Martensstraße 1/3
D-W 8520 Erlangen, Germany

Abstract. In this paper the MEMSY experimental multiprocessor system is described. This system was built to validate the MEMSY architecture - a scalable multiprocessor architecture based on locally shared-memory and other communication media. It also serves as a study of different kinds of application programs which solve a variety of real problems encountered in scientific research.

1 Introduction

Among the different kinds of multiprocessor systems, those with global shared-memory are normally the ones most liked by application programmers because of their simple programming model. Closer examination of typical problems reveals that, for a broad class of these problems, global shared-memory is not what is really needed by the application. Local shared data is sufficient to solve the problems.

Multiprocessors with global shared-memory all suffer from a lack of scalability. By making clever use of fast buses and caching techniques this effect may be postponed, but each system has an upper limit on the number of processing nodes.

Our MEMSY system is now an approach towards a scalable MIMD multiprocessor architecture which utilizes memory shared between a set of adjacent nodes as a communication medium. We refer to this kind of shared-memory as *distributed shared-memory.*

The MEMSY system shall continue the line of systems which have been built at Erlangen using distributed shared-memory. The basic aspects of this architecture are described in [2] and [3].

2 MEMSY Architecture

2.1 Design Goals

The MEMSY architecture was defined with the following design goals in mind:

- economy The system should be based on off-the-shelf components we can buy; only some parts should need to be designed and built by us.

- scalability The architecture should be scalable with no theoretical limit. The communication network should grow with the number of processing elements in order to accommodate the increased communication demands in larger systems.
- flexibility The architecture should be usable for a great variety of user problems.
- efficiency The system should be based on state-of-the-art high performance microprocessors. The computing power of the system should be big enough to handle real problems which occur in scientific research.

2.2 Topology of the MEMSY System

The MEMSY structure consists of two planes. In each plane the processor nodes form a rectangular grid. Each processor node has an associated shared-memory module, which is shared with its four neighbouring processor nodes. The grid is closed to a torus.

One processing element of the upper plane has access to the shared-memory modules of the four processing elements directly below it, thereby forming a small pyramid. There are four times as many processing elements in the lower plane than in the upper.

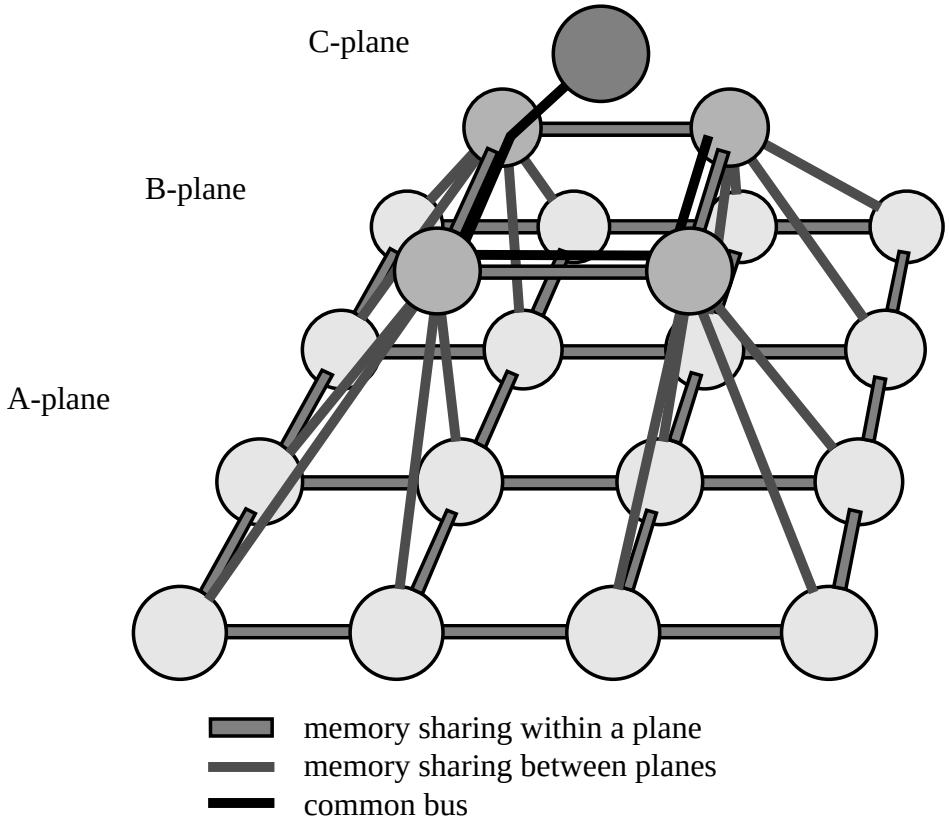


Fig. 2.1 Topology of the MEMSY system (Torus connections are missing)

On top of the whole system there is an optional processor which may serve as a front-end to the system.

The basic idea behind this structure is this: the lower plane does the real work and the upper plane feeds the lower plane with data and offers support functions.

A subset of the processor nodes has access to one or more common bus systems to allow communication over greater distances or broadcasts.

3 Hardware Architecture of MEMSY

The experimental memory-coupled multiprocessor system MEMSY consists of three functional units:

- 4 + 16 processor nodes,
- one shared-memory module at each node, which is called the *communication memory* and
- the interconnection network which provides the communication paths between processor nodes and communication memories.

In addition to these essential units which are described in the following sections, an FDDI net, a special optical bus and a distributed interrupt coupling system are integrated in the system. The FDDI net allows testing and use of another communication media. The optical bus is designed to support various global synchronisation mechanisms. The interrupt coupling system establishes interrupt connections between every two immediate-neighbour nodes which share a communication memory.

3.1 Processor Nodes

Each node of the MEMSY system is identically designed and consists of a Motorola multiprocessor board MVME188 and additional hardware, some of which were designed and implemented as part of the MEMSY project. In figure 3.1, which shows the logical node structure, these parts have a lighter background colour.

The MVME188 board used in the MEMSY system consists of four VME modules. These are; the system controller board, holding e.g. timers and serial interfaces; two memory boards, each holding 16M bytes local memory; and the main logic board, carrying the processor module.

The processor module comprises four MC88100 RISC CPUs, which have multiple internal parallelism, and eight MC88200 cache and memory management units (CMMU), which provide 8*64K bytes cache memory. Special features of the processor module are:

- Cache coherency is supported by the hardware.
- There exists a cache copyback mode which writes data back to memory only if necessary and a write-through mode.
- There exists an atomic memory access which is necessary for efficiently implementing spinlocks and semaphores in a multiprocessor environment.
- The caches provide a burst mode which allows atomic read/write access of four consecutive words[1] while supplying only one address.

The architecture of the processor module, the MVME188 board and the M88000 RISC product[2] are described in greater detail in [9].

All VME modules mentioned above, are interconnected by a high speed local bus. Each node can be extended with additional modules via this local bus or the VME bus. The communication memory interface is attached to the local bus. Its function is to recognize and execute accesses to the communication memories. The interface hardware provides three ports to which the communication memories are connected either directly or via an interconnection network. This interconnection network is described in section 3.2.

To the M88000 the memory interface looks like a simple memory module. The address decoder of the MVME188 is configured in such way, that the highest two address bits determine whether the address space of the local memory boards, of the VME bus or of the memory interface is accessed. In case of the memory interface, the next two address bits determine the port which should be used for each memory access. Four further bits of the address determine the path through the coupling unit and which communication memory is to be accessed.

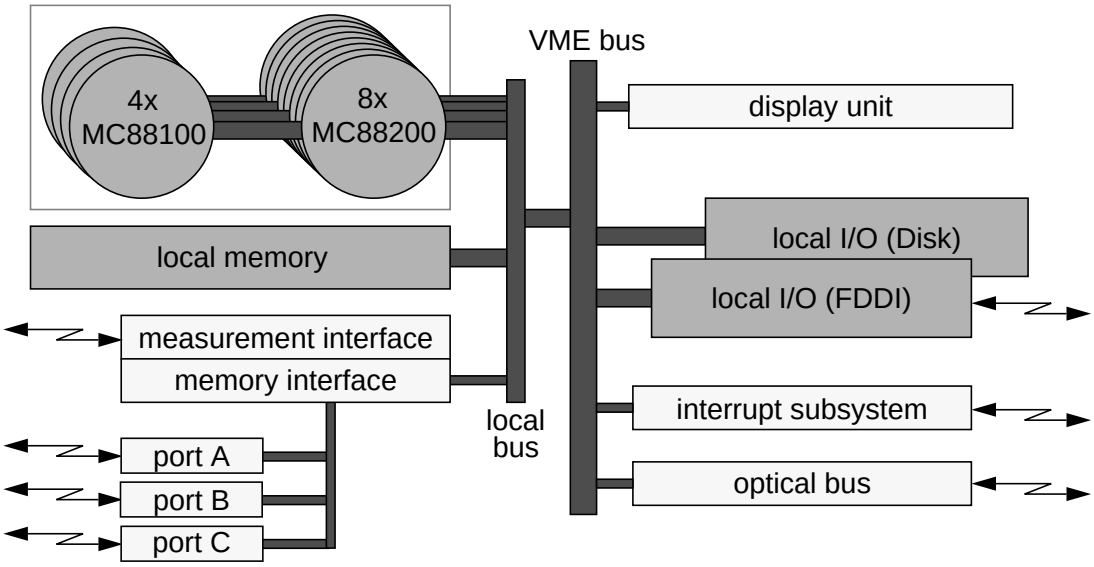


Fig. 3.1 Logical node structure

Addresses and data are transferred in multiplexed mode. The connection is 32 data bits plus four parity bits wide for the address or data transfer. The parity bits are generated by the sender and checked by the receiver. If the communication memory detects a parity error in address or data, it generates an error signal, otherwise a ready signal is generated. If the memory interface receives an error signal or detects a parity error during a read access it transmits an error signal to the M88000, otherwise a ready signal is sent.

---

1. A word is always 32 bit wide.
2. The combination of MC88100 and MC88200 is referred to as the M88000 RISC product.

The memory interface hardware supports the atomic memory access and the described burst mode. Counters have been included in the interface hardware to count the various types of errors in order to investigate the reliability of the connection. The counters can be read and reset by a processor. In addition there is a status register which contains information about the last error occurred. This can be used in combination with an error address register to investigate the error.

In addition, the memory interface contains a measurement interface to which an external monitor can be connected. A measurement signal is triggered by a write access to a particular register of the memory interface and the 32-bit word written is transferred to the monitor. This enables the use of hybrid monitors for measurements on MEMSY. Refer to [5] for further information on this topic.

3.2 The Interconnection Network

The topology of MEMSY is described in section 2.2. Each node has access to its own communication memory and to the communication memories of the four neighbouring nodes. Additionally, every node of the B-plane has access to the communication memories of its four assigned A-plane-nodes.

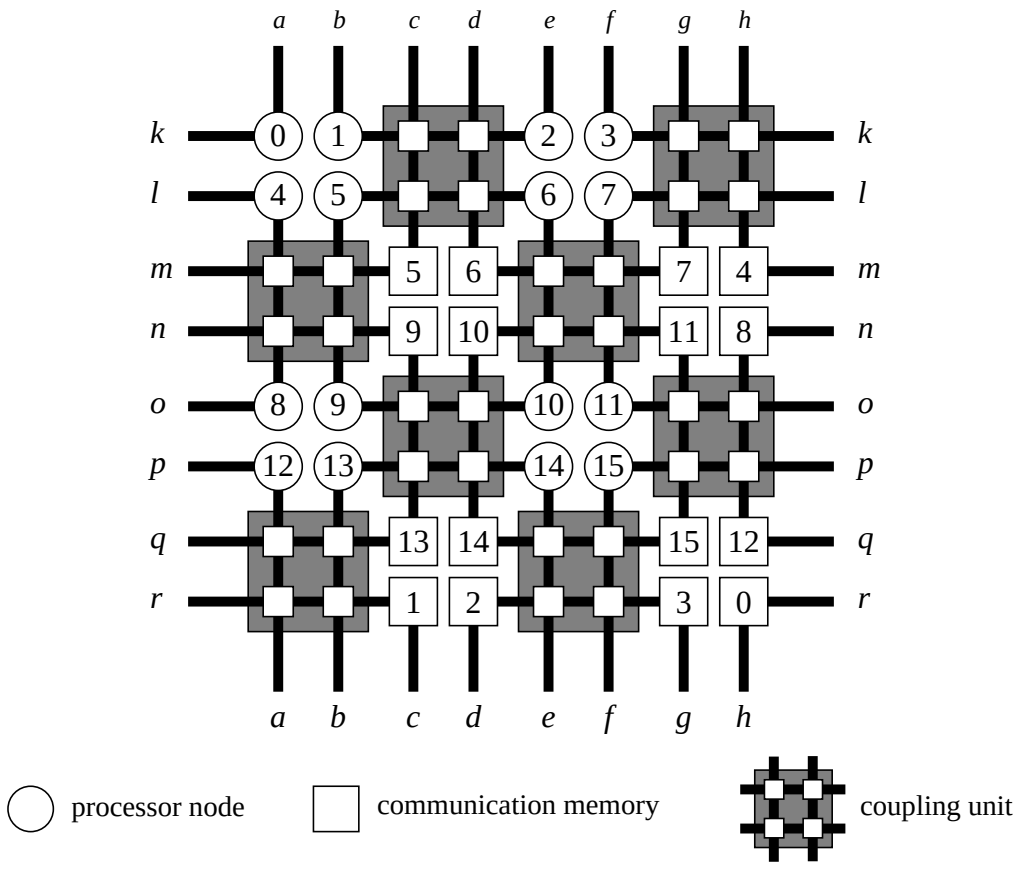


Fig. 3.2 Composition of processor nodes, communication memories and coupling units

A static implementation of this topology requires up to 9 ports at each node and up to 6 ports at each communication memory. To reduce this complexity and the number of interconnections, a dynamic network component, called *coupling unit*, has been developed. The use of the coupling unit reduces the number of ports needed at the memory interface and the communication memory to three. Only two of these ports are used for the connections within a plane. The coupling unit supports the virtual implementation of the described MEMSY topology.

The coupling unit is a blocking, multistage, dynamic network with fixed size which provides logically complete interconnection between 4 input ports and 4 output ports. The interconnection structure of MEMSY is a hybrid network with global static and local dynamic network properties.

The torus topology of a single MEMSY plane is implemented by the arrangement of nodes, communication memories, and coupling units as shown in figure 3.2. For reasons of complexity the local dynamic network component is not depicted in this figure. It is described in more detail in the next section.

Each node and each memory module is connected to two coupling units. Thus the nearest-neighbour torus topology can easily be established. A square torus network with $N=n^2$ nodes requires $N/2$ coupling units. The connections from the nodes of the B-plane to the four corresponding communication memories of the A-plane are also implemented by using coupling units. These are connected to the third ports.

Internal Structure of the Coupling Unit. The hardware component used to implement a multiprocessor system, as described above, is shown in figure 3.3. In our implementation of the interconnection network, accesses to the communication memories via coupling units are executed with a simple memory access protocol. The interconnection network operates in a circuit-switching mode by building up a direct path for each memory access between a node and a communication memory.

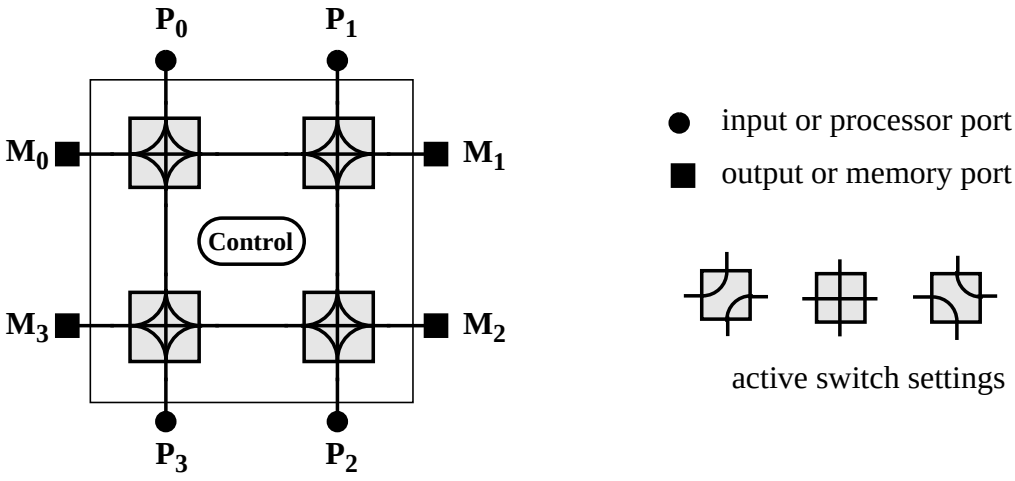


Fig. 3.3 Internal structure of a coupling unit

A coupling unit consists of the following subcomponents:

- 4 p-ports which allow the access to the coupling unit from nodes
- 4 m-ports which provide the connection of communication memories
- 4 internal subpaths which perform data transfer within the coupling unit
- 1 control unit which controls the dynamic interconnection between p-ports and m-ports
- 4 switching elements which provide the dynamic interconnection of p-ports and m-ports

The structure of the p-ports and m-ports is basically identical to a memory interface with a multiplexed 32 bit address / data bus. The direction of the control flow is different for p-ports and m-ports. An activity (a memory access) can be only initiated at a p-port.

The control unit is a central component within the coupling unit. It always has the complete information about the current switch settings of all switching elements. If a new request is recognized by receiving a valid address, the control unit can decide at once whether the requested access can be performed or has to be delayed. For any access pattern the addressed memory port and all necessary internal subpaths are available when all switching elements contained in the communication path to be built-up are either inactive or possess exactly the switch settings required for the establishment of the interconnection.

The necessary switch settings of all required switching elements are fixed a priori for every possible access pattern. The decision about the performability of a requested access is made by comparing the required switch settings with the current ones.

It can be seen from the structure of the coupling unit that two different communication paths containing disjoint sets of internal subpaths can be selected for a memory access. This results from the arrangement of the switching elements in a ring configuration interconnected by the internal subpaths. The decision as to which of the possible communication paths is to be established is made dynamically according to the current switch settings. If possible, the communication path requiring fewer internal subpaths is chosen to minimize the propagation delay caused by the switching elements.

The feature of the coupling unit which allows alternative communication paths is important in the context of fault tolerance. This is discussed in [1].

Performance of the Interconnection Network. An access to shared data in the communication memories requires a significantly higher access time than an access within the node. In addition to the fact that the reduction in access time caused by using a cache is generally no longer possible, the longer transfer paths and the execution of control mechanisms cause further delays. The sequentialization which can be required when conflicts occur either in the communication memories or in the coupling units can cause additional waiting times. The memory access time in our implementation is normally 1 μs and up to 1.3 μs if blocking occurs due to a quasi-simultaneous access.

Since only data which is shared by nodes is held in the communication memories, such as boundary values of subarrays, the increased access time has only a small influence on the overall computing time. Measurements made using the test system INES [4] specially developed to measure the performance of the coupling hardware show that a high efficiency can be achieved under realistic conditions. Thus reducing the complexity of the network by using coupling units causes only a small reduction in performance compared to a static point to point network.

4 Programming Model

The programming model of the MEMSY system was designed to give the application programmer direct access to the power of the system. Unlike in many systems, where the programmer's concept of the system is different from the real structure of the hardware, the application programmer for MEMSY should have a concept of the system which is very close to its real structure. In our opinion this enables the programmer to write highly efficient programs which make the best use of the system.

In addition, the programmer should not be forced to a single way of using the system. Instead, the programming model defines a variety of different mechanisms for communication and coordination[3]. From these mechanisms the application programmer may pick the ones which are best suited for his particular problem.

The programming model is defined as a set of library calls which can be called from C and C++. We choose these languages for the following reasons:

(1) Only languages which have some kind of a 'pointer' make it possible to implement the routines, which access the shared-memory, as library calls. Otherwise costly extensions to the languages would have been needed.
(2) The C compiler is available on every UNIX system. As it is also used to develop the operating system itself, more effort is taken by the manufacturer to make this compiler bug-free, stable and have it generate optimized code.
(3) Compared to programs using index references, programs using pointer references can lead to more efficient machine code.

The MEMSY system allows different applications to run simultaneously. The operating system shields the different applications from one another.

To make use of the parallelism of each processing unit, the programmer must generate multiple processes by the means of the 'fork' system call.

4.1 Mechanisms

The following sections introduce the different mechanisms provided by the programming model.

3. We use the term *coordination* instead of *synchronization* to express that not the simultaneous occurring of events (e.g. accesses to common data structures) is meant but their controlled ordering.

Shared-Memory. The use of the shared-memory is based on the concept of ‘segments’, very much like the original shared-memory mechanism provided by UNIX System V. A process which wants to share data with another process (possibly on another node) first has to create a shared-memory segment of the needed size. To have the operating system select the correct location for this memory segment, the process has to specify with which neighbouring nodes this segment needs to be shared.

After the segment has been created, other processes may map the same segment into their address space by the means of an ‘attach’ operation. The addresses in these address spaces are totally unrelated, so pointers may not be passed between different processes. The segments may also be unmapped and destroyed dynamically.

There is one disadvantage to the shared-memory implementation on the MEMSY system. To ensure a consistent view over all nodes the caches of the processors must be disabled for accesses to the shared-memory. But the application programmer may enable the caches for a single segment if he is sure that inconsistencies between the caches on different nodes are not possible for a certain time period. The inconsistencies are not possible if only one node is using this segment or if this segment is only being read.

Messages. There are two different message mechanisms which are offered by the programming model: the one described in this section and the one named ‘transport’, described later.

This message mechanism allows the programmer to send short (2 word) messages to another processor. The messages are buffered at the receiving side and can be received either blocking or non-blocking. They are mainly used for coordination. They are not especially optimized for high-volume data transfer.

Semaphores. To provide a simple method for global coordination, semaphores have been added to the programming model. They reside on the node on which they have been created, but can be accessed uniformly throughout the whole system.

Spinlocks. Spinlocks are coordination variables which reside in shared-memory segments. They can be used to guard short critical sections. In contrast to the other mechanisms this is implemented totally in user-context using the special machine instruction ‘XMEM’. The main disadvantage of the spinlocks is the ‘busy-wait’ performed by the processor. This occurs if the process fails to obtain the lock and must wait for the lock to become free. To minimize the effects of programming errors on other applications, a time-out must be specified, after which the application is terminated (there is a system-imposed maximum for this time-out).

Transport. The transport mechanism was designed to allow for high volume and fast data transfer between any two processors in the system. The operating system is free to choose the method and the path this data is to be transferred on (using shared-memory, FDDI-ring or bus). It can take into account the current load of the processing elements and data paths.

I/O. Traditional UNIX-I/O is supported. Each processing element has a local data storage area. There is one global data storage area which is common to all processing nodes.

Parallelism. To express parallelism the programmer has to create multiple processes on each processing element by a special variant of the system call ‘fork’.

Parallelism between nodes is handled by the configuration to be defined: One initial process is started by the application environment on each node that the application should run on. In the current implementation these processes are identical on all nodes.

Information. The processes can obtain various information from the system regarding their positions in the whole system and the state of their own or other nodes.

4.2 Development

The programming model is open for extensions which will be based on experiences we gain from real applications. Specific problems will show whether additional mechanisms for communication and coordination are needed and how they should be defined.

5 Operating System Architecture

Various considerations have been made as to which operating system should be chosen. Basically there are two choices:

- Design and implement a completely new operating system or
- use an existing operating system and adapt it to the new hardware.

By designing a new operating system the whole hardware can be completely integrated and supported. Better fitting concepts than those found in existing implementations can be developed. But it must not be underestimated, that implementing a new operating system requires a lot of time and effort. The second choice offers a nearly ready-to-run operating system, in which only the adaptions to the additional hardware have to be made.

For MEMSY the second choice was taken and Unix was chosen as the basis for MEMSOS, the operating system of MEMSY. Unix supplies a good development environment and many useful tools. The multitasking / multiuser feature of Unix is included with no additional effort.

On each processor node we use the UNIX SYSTEM V/88 Release 3 of MOTOROLA, which is adapted to the multiprocessor architecture of the processor board. The operating system has a peer processor architecture, meaning that there is no special designated processor, e.g. master processor. Every processor is able to execute user code and can handle all I/O requests by itself. The kernel is divided into two areas. One area contains code that can be accessed in parallel by all processors, because there is either no shared data involved or the mutual exclusion is achieved by using fine grain locks. The second area contains all the other code that can not be accessed in parallel. This area is secured with a single semaphore. For example, all device drivers can be found here. In SYSTEM V/88 the usual multiprocessor concepts are implemented, such as message-passing, shared-memory, interprocessor communication and global semaphores. See [8] for more details.

5.1 Extensions

For the implementations of the above mentioned multiprocessor concepts the assumption has been made that all processors share a global main memory. But the operating system is not able to deal with distributed memory such as our communication memory. Therefore certain extensions and additions have been made to the operating system. Only little changes have been made to the kernel itself. Standard Unix applications are runnable on MEMSY because the system-call interface stayed intact.

Integration of the additional hardware, particularly the communication memories and the distributed interrupt-system, was one of the first steps. One of the next steps made was the implementation of basic mechanisms for all sorts of communication and coordination, which depend on the shared-memory. On top of these mechanisms most of our other extensions are built. The Unix system-call interface was extended by additional system calls, as described in section 4.1.

In the following sections only some of the extensions are described. Our concept for support of user programs and a hierarchy of communication mechanisms using the distributed shared-memory is introduced.

Support of Distributed User Programs. Various demands on high-performance multiprocessor systems are made by the users. A system should be highly available and easy to use. There should be as little interference with other user programs as possible and the computing power should always be the maximum available. For their programs users demand the usual, or even an enlarged functionality, short start-up times and the possibility of interactive testing.

The Application Concept. In MEMSOS most of the users' needs are supported by the realization of our *application concept*. We define the set of all processes belonging to one single user program as an *application*. An application can be identified by a unique *application number*. Different applications running on MEMSY are distinguishable by that number. Single application processes, called *tasks*, inherit the

application number and are assigned a *task number*, which is a serial number unique for this application and processor node. So an application task can be identified by its application number, task number and node number.

At system initialization time one outstanding application is created. This initial application, the *master application*, is made up of application daemons running on each node and a single master application daemon, which may be distributed. All daemons communicate with each other. It is the purpose of these daemons to create the user applications on demand and to keep track of them. For each new application the master daemon allocates a free, unique application number. As the first task of each application the application *leader task* is started on each node by the other daemons. The leader task creates the application environment and all subsequent tasks. It supervises the execution of the application tasks on that node it is running on and communicates with all leader tasks of the same application. The leader tasks act as agents to the master application.

This simple application concept makes it possible to easily control and monitor distributed user programs. Because single applications can be distinguished from one another, more than one application can be allowed to run in parallel on MEMSY. By changing the processor binding and process/task scheduling more efficiency can be achieved.

Processor Binding and Process Scheduling. In the SYSTEM V/88 operating system each processor can have its own assigned process run queue which it prefers to use. Each Unix process is bound to a certain run queue. During a process switch the binding can be changed. The run queue, bound to a processor, can also be changed depending on system work load. In the original implementation (R32V3.0) there was only one run queue for all processors assigned, although more run queues could have been possible. See [7] for more details.

To support applications more efficiently the processor binding and the number of run queues was altered. In our implementation there exists one *system run queue* for all system processes, which is bound to one of four processors. For each application running on a node an *application run queue*, local to that node, will be created. These application run queues are handled by the remaining processors. The binding is not static and can be changed dynamically.

Additionally a new concept called *gang scheduling* has been realized. Each application is assigned an application priority. The tasks of the application with the highest priority will be scheduled first. The application priority can change dynamically, depending on the system work load. The system workload is supervised by a distributed scheduler process, which will change application priorities and processor binding accordingly.

Interrupt Mechanism. As shown in section 3.2 the access time to the communication memory is higher than, for example,. to the local memory. To use polling mechanisms on the communication memory is therefore very inefficient and must be avoided, at least in the kernel. Because of this an interrupt connection between

nodes is necessary. This connection is made only between those immediate nodes which share a communication memory. We use a special hardware, supported by software, to generate these inter-node interrupts.

With every interrupt triggered a word is provided at a defined memory location in the communication memory owned[4] by the triggering node. The interrupt hardware recognizes the port on which the interrupt occurred and the software can locate the corresponding node number and communication memory, from which the supplied word can be read.

The interrupt word consists of two parts. The first part is 8 bit wide and represents the *interrupt type*, the second part, the *data-part*, is 24 bit wide and is free for other use. The interpretation of the type depends on the data-part.

An interface is provided by the interrupt module, so that it can be used by other kernel modules[5]. A module has to reserve interrupt types as needed and register a callback function for every reserved type. Some types are already reserved by the interrupt module itself. They are used e. g. for establishing an initial connection or for state information important for other nodes. For a kernel module the reserved types must be system-wide identical. A single *interrupt-send* routine is provided to initiate an interrupt. Parameters of this function are the destination node number, the interrupt type, the data-part and a time-out value. With this time-out value one can switch between time-out mode, blocking and non-blocking mode.

In case of an interrupt the interrupt mechanism reads the supplied interrupt word, extracts the type and then calls the registered callback function with the senders node number, the interrupt type and the data-part as parameters.

Certain enhancements have been implemented to increase the robustness of the interrupt mechanism. The interrupt mechanism automatically tries to establish connections to all immediate neighbours which are accessible. It monitors these connections and reports changes in status to the kernel modules by using the callback functions. By using a FIFO queue, the interrupt mechanism is able to smooth short peaks in the interrupt frequency.

Message-Passing Mechanism. The message-passing mechanism was implemented as one of those kernel modules using the interrupt mechanism.

A message consists of the message header and the message body, which can hold six words (24 bytes). For the messages a static buffer pool is allocated in the communication memory modules which the node owns. A special buffer management was implemented. It has the responsibility of keeping track of each buffer sent. This is very important for maintaining consistency of the buffer pool.

The interface of message-passing module is constructed in the same way as the interface of the interrupt module. One has to reserve message types and register a callback function for each type reserved. To actually send a message a single *message-send* function is provided.

---

4. To provide a uniform structure of the communication memory, the physical memory may be divided into logical parts, which may be assigned to different nodes.
5. We call each of our implemented kernel extensions a module.

If the *message-send* routine is called, the message-passing mechanism allocates a buffer for the message, fills in the message header and copies the message body. Some processor nodes may have more than one logical communication memory, so the buffer is allocated in that memory module which the receiver or the routing node has access to. The message-passing mechanism then calls the *interrupt-send* function with parameters destination node, type and index of the allocated message buffer.

Message buffers sent to an immediate neighbour are not sent back immediately, but are gathered by the receiver. This is done to reduce the interrupt rate. There are three events in which accumulated buffers are sent back:

- A certain amount is exceeded. The limit is a tunable parameter.
- A message is sent in the opposite direction. The accumulated buffers belonging to the receiver are simply added to the message.
- A neighbour requests the return of the used buffers.

A simple protocol guarantees that a message is received by the destination node. Additional protocols assure that a received message is accepted by the destination kernel module.

Shared-Memory Mechanism. Another communication mechanism, beside the message-passing mechanism, is the shared-memory mechanism. In the following section we introduce our implementation of this mechanism.

The shared-memory mechanism consists of two parts. These are the communication memory manager which provides the linkage to the physical shared-memory and, as main part, the shared-memory manager. The shared-memory manager implements the necessary protocols to maintain the consistency of allocated shared-memory segments. It also supplies a simple interface useable by other kernel modules.

Communication Memory Manager. To allocate or free pages[6] from the communication memory, the shared-memory mechanism uses calls to the communication memory manager. The pages available for shared-memory are numbered consecutively and linked by using a table containing one entry for each page. The entries determine the number of the following page, which need not be the one physically following. What distinguishes this memory manager from others is the lack of automatic memory mapping or unmapping. Therefore a call to the *allocate* function does not return the start address of the allocated memory, but a pointer to a table containing the page numbers. The information about the pages is essential, because the address space mapping may not be the same on all nodes. All tables are situated in the communication memory itself so that they are accessible by immediate-neighbour nodes. Additional calls exist for calculating addresses out of page numbers and for mapping and unmapping the allocated memory into and out of the kernel address space.

6. The hardware only supports pages of 4K byte granularity.

Shared-Memory Manager. The shared-memory manager provides the functionality used for communicating with immediate neighbours and keeps track of allocated pages to allow their re-integration in case of faults on neighbouring nodes. On allocation of a shared-memory segment, the memory manager validates provided parameters and chooses that communication memory which the destination node has access to. If an immediate neighbour wants to share an allocated memory segment, the memory manager provides upon request the offset to the corresponding page table. For the inter-node communication the message-passing mechanism is used. On the destination node the shared-memory manager is able to locate the page table and to map the shared segment into the kernel address space.

On top of the shared-memory manager a system-call interface (as described in [6]) is established. This interface allows an efficient use of the shared-memory mechanism by the application programmer.

In MEMSOS we want to examine certain aspects of different communication mechanisms. In section “Message-Passing Mechanism” the basic message-passing mechanism was introduced. This mechanism was built on top of the interrupt mechanism described above. In this section we have described the shared-memory mechanism. In this shared-memory mechanism we use the message-passing mechanism as the basis for communication. This was done because the amount of communication needed and the time used for it is fairly small in comparison with the data transferred and the time needed for reading and writing the data.

6 Conclusions

In this paper the MEMSY project was introduced. MEMSY belongs to the class of the shared-memory multiprocessors. Viewed from the hardware level massive parallel systems with global shared-memory have not been realizable up to now. A compromise has to be made between an efficient system on one side and a general purpose system on the other side.

MEMSY offers tightly coupled processor nodes arranged in a hierarchy of grids. The sharing of memory is only between nearest-neighbour nodes. It was shown that with the additional hardware, called coupling units, it is possible to reduce the amount of necessary connections without too much loss in efficiency. Because of the constant complexity of inter-connections and the modular concept the system is easily scalable.

An easy to use programming model which offers even a great variety of paradigms used in the programming models of other high-performance multiprocessor systems was introduced. Because of this programming model many existing user programs are easily portable to our system. Currently some programs are ported to the MEMSY system. By examining these programs we hope to gain more information about the system performance and be able to take valid measurements.

Our application concept was introduced. It offers a way to supervise distributed user programs. It is the basis for further work to be done in the area of user support and load balancing.

Because MEMSY is an experimental multiprocessor system it was tried to implement as many communication mechanisms as possible. An important aspect in doing this was to be able to compare their usefulness and performance and therefore be able to validate our multiprocessor concept.

References

1. M. Dal Cin et al., “Fault Tolerance in Memory Coupled Multiprocessors”; in this volume
2. G. Fritsch et al., “Distributed Shared-Memory Architecture MEMSY for High Performance Parallel Computations”; *Computer Architecture News*, Vol. 17, No. 6, Dec. 1989, pp. 22 - 35
3. W. Händler, F. Hofmann, H.-J. Schneider, “A General Purpose Array with a Broad Spectrum of Applications”; *Computer Architecture*, Informatik Fachberichte, Springer Verlag, No. 4, 1976, pp. 311-335
4. U. Hildebrand, *Konzeption, Bewertung und Realisierung einer dynamischen Netzwerkkomponente für speichergekoppelte Multiprozessoren*, Dissertation, Arbeitsberichte des IMMD, Univ. Erlangen-Nürnberg, Band 25, No. 5, 1992
5. R. Hofmann, “The Distributed Hardware Monitor ZM4 and its Interface to MEMSY”; in this volume
6. Kardel, W. Stukenbrock, T. Thiel, S. Turowski, *Anleitung zur Benutzung des MEMSY-Programmiermodells für Anwender*; interner Bericht, IMMD 4, Univ. Erlangen-Nürnberg, Oktober 1991
7. Karl J. Rusnock, *Multiprocessor SYSTEM V/88 Release 3 Design Specification*; Motorola, Confidential Proprietary, May 1989
8. K. Rusnock, P. Raynoha, “Adapting the Unix operating system to run on a tightly coupled multiprocessor system”; *VMEbus Systems*, Oct. 1990, Vol. 6, No. 5, pp. 8-28
9. K. Rusnock, *The Multiprocessor M88000 RISC Product*; Motorola Microcomputer Division, Tempe, AZ 85282, 1991