MEMSY
A Modular Expandable Multiprocessor System with Fault Tolerance

W. Hohl, M. Dal Cin, J. Hönig, A. Pataricza[1]

University of Erlangen-Nürnberg
IMMD III, Martensstraße 3, 91058 Erlangen, FRG
hohl@informatik.uni-erlangen.de

Abstract

The experimental multiprocessor system MEMSY[2] will be described. This system was built to validate the concept of the MEMSY architecture - a scalable multiprocessor architecture based on local shared-memory. Main application areas are scientific computations with a high demand for processing power and memory capacity. In designing the hardware architecture the extensive use of standard components was a condition. The programming model of MEMSY is custom made reflecting its real hardware structure whereas th eoperating system is a Unix extension. In massively parallel systems with its complexity and large number of components the chance of a single or multiple failure is no longer negligible. It is clear that the redundancy, reconfigurability and diagnosis techniques must be incorporated at the design stage itself and not as a subsequent add-on.

Keywords: Multiprocessor, MIMD, Scalability, Fault Tolerance

1. Introduction: Motivation and Design Goals

There are some well known reasons to develop massively parallel multiprocessors [2], [9]as:

- the demand for increasing amounts of computing power and memory capacity, for example to attack the “Grand Challenges” [7]
- the evolution in logic technolgy to very large scale integration connected with decreased cost, increased speed and a dramatic reduction in power requirements
- the insight that a broad spectrum of applications is unsuitable to vector processors
- extremely better price/performance ratio of RISC to vector processors
- technological limitations for the mainframes.

Among the different kinds of parallel computers the systems of the MIMD-type are especially versatile usable due to its independent instruction streams which may compute the same or different data streams in the same or a different manner. In the class of multiprocessor systems, those with global shared-memory are normally favorite of application programmers because of its simple programming model.

Main application areas are scientific computations with a high demand for processing power and memory capacity, such as numerical simulation of physical phenomena or quantum mechanical computations. Analysis of typical problems reveals that a broad class of these problems is regularly structured and can easily be partitioned and distributed to a regularly structured multiprocessor. Therefore, global shared-memory is not what is really needed by the application. Local shared data is sufficient to solve the problems.

1. Guest researcher from TU Budapest, Dept. Measurement and Instrumentation Engineering

2. Supported by the DFG (Deutsche Forschungsgemeinschaft) as part of the “Sonderforschungsbereich” SFB 182

Besides, 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. We believe that our kind of shared-memory called *distributed shared-memory* is more efficient than virtual shared-memory implementations and so we decided not to build a global shared-memory or virtual shared-memory architecture.

Massively parallel systems represent a new challenge for fault tolerance. The designers of such systems cannot expect that no parts of the system will fail. With the significant increase in the complexity and number of components the chance of a single or multiple failure is no longer negligible. It is clear that the redundancy, reconfigurability and diagnosis techniques must be incorporated not as a subsequent add-on but at the design stage itself.

The MEMSY architecture was defined with the following design goals in mind [10]:

- Economy: The system should be almost completely based on off-the-shelf components.
- 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 high enough to handle problems of scientific research.
- Fault-Tolerance: The system should provide long uninterrupted computation times and reliable access to a large amount of data over a long time span.

In the following we give an overview on the realization of these ideas. In Section 2 the hardware architecture of MEMSY [10] is described. In Section 3 we present the programming model and the operating system. Section 4 describes the fault treatment.

2. Hardware Architecture

The MEMSY architecture consists of tightly coupled processor nodes arranged in a two-level hierarchy of four-neighbour toroidal grids. The processors of nearest-neighbouring nodes communicate through shared memories (called *communication memories*) . All of the nodes have the same structure.

2.1. Topology of MEMSY

The MEMSY structure consists of two levels corresponding to their different tasks: Applications at the A-level resp. communication with the front-end and system control at the B-level (Fig. 1). In each level the processor nodes form a rectangular grid. Each processor node has an associated communication memory, 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.

This topology offers constant local interconnection complexity, one important condition for scalable systems.

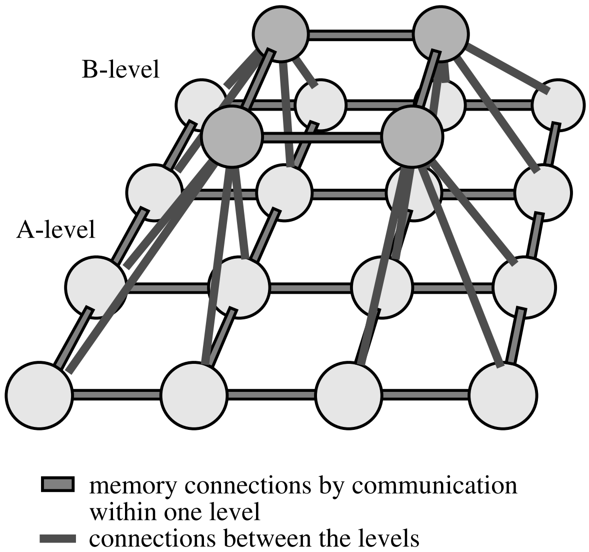

Fig. 1 MEMSY topology

We have built an experimental system consisting of three functional units:

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

2.2. Processor Nodes

Each of the nodes of the pilot system is based on the Motorola MVME 188 board system supplemented by additional hardware, some of which were designed and implemented as part of the MEMSY project. In Fig 2, which shows the logical node structure, these parts have a lighter background colour.

The MVME188 board used consists of four VME modules: The system controller board, holding e.g. timers and serial interfaces; two memory boards, each holding 16 MB local memory; and the main logic board, carrying the processor modules.

The processor module comprises four MC88100 RISC CPUs and eight MC88200 cache and memory management units (CMMU), which provide 8*64 KB cache. Internally the processors have a Harvard-architecture, i.e. each one has a separate data and instruction bus.

Special features of a processor module are [19]:

- Cache coherency is supported by 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 while supplying only one address.

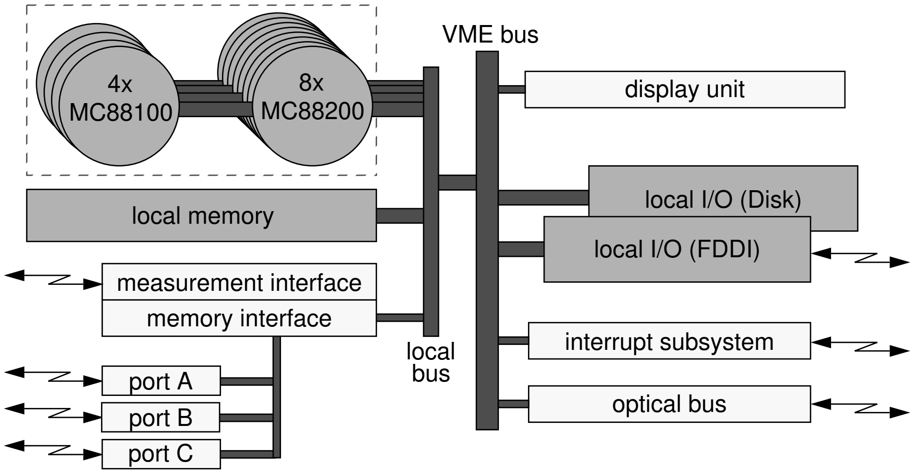

Fig. 2 Logical node structure

All VME modules mentioned above, are interconnected by a high speed local bus. Each node can be extended by additional modules via this local bus or the VME bus as e.g. by a watchdog processor (described in section 2.2.2.2.). 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 (described in section 2.4.).

To the processor node the memory interface looks like a simple memory module. The address decoder of the MVME188 is configured in such a 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.

Addresses and data are transferred in multiplexed mode. The connection is 32 data bits plus four parity bits wide . 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 the error signal to the processor, otherwise a ready signal is sent.

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.

2.3. Error Detection on System Level

The primary goal in designing error detection mechanisms is high fault coverage, while keeping redundancy on a moderate level. This can be achieved by combining built-in standard mechanisms on chip level (see above) with efficient hardware fault detection mechanisms on system level.

For economic reasons, the large hardware overhead resulting from multiple modular redundancy for the whole system is not feasible. Redundancy has to be restricted to the most crucial hardware resources, such as the CPU. Duplication within the computing nodes can be based on master-checker mode or on the use of a watchdog processor to monitor concurrently the program execution of the main processor. A consequence of our experience is, that the cumulative effect resulting from the simultaneous use of both methods qualitatively exceeds the simple superposition of the individual effects [5].

2.3.1. Master-Checker Mode

The master-checker (MC) mode is based on the duplication of processors. Both processors run the same program clock-synchronously and process the same data stream. In the Motorola M88k processor family, MC-mode can be set-up practically without external components, as the necessary logic is integrated onto the chip itself. An internal comparator is assigned to each bus pin, which compares internal and external values at the output pins. Only one processor (the master) outputs data. The output bus drivers of the other processor (the checker) are disabled. Accordingly, during data transfer the comparators on the checker pins compare the internal signals and those driven by the master. In the case of mismatch an error is signalled.

This mode is implemented in an experimental version of the MEMSY node by use of the Motorola MVME 188 board system comprising two identical MC88100 processors. Both processors can either be used independently or jointly with reduced performance as MC-system. In MC-mode the corresponding pins of both integrated circuits (CPU and/or MMU chips) are interconnected by means of an add-on board.

Duplication ensures full coverage of transient and permanent faults in a single unit of a MC-pair, but it does not detect errors affecting both units, e.g an error in the instruction or data stream. But, combined with parity protection of memories and busses, the error detection of the computing core covers most single faults.

In addition to fault coverage, short error latency is a primary objective as well. A long error latency may result not only in a significant loss of computing time, but can also cause serious difficulties in fault diagnosis, since long fault propagation chains weaken the correlation between a fault and the resulting operational errors.

In MC-mode, only the data transfer on the bus can be observed; data in the CPU and in the cache remain unchecked until they are written back to the main memory. The Motorola CMMU supports three different memory update policies: *cache inhibit*, *write-through* and *copy-back*.

For the most frequently used (and probably most important) data copy-back to the main memory can be delayed until the very end of the computation. Therefore, with copy-back the error latency can, in the worst case, approach the full computing time! In Table 1 the fault coverage of MC-based checks of different bus transactions are summarized for different memory update policies.

For the instruction fetch only read operations are performed. Hence, there are only two alternative cache policies: cache enabled and cache inhibited. Only if the cache is inhibited, the MC-setup performs checks on the instruction flow with low latency. However, inhibiting the cache may reduce performance drastically.

The most important limitation of the error detecting capabilities of the MC-mode results from the only partial duplication, i.e. the system remains still unprotected against faults in common resources located in the "outside world" of the processing core. Hence, in addition to the MC-mode of the CPU an error detection mechanism is required, which checks the instruction flow and guarantees at least a moderate fault coverage for the whole system.

2.3.2. Watchdog Processors for Control-Flow Checking

A correct control-flow is a precondition for correct program execution. Every event resulting in a failure in the content of the program counter (e.g. changes in the return address after a subprogram has been called, or a failure in the memory or in the address counter) can cause a control-flow error. For concurrently checking the program control-flow a watchdog processor (WDP) [17] can be attached to each main processor (or to a group of them).

A WDP, realized as a relatively simple coprocessor, compares previously computed reference signatures with run-time signatures, which encode some compacted characteristics of the reference and actual program flow. At program start the reference signatures are transferred to the WDP. The run-time signatures are generated by the WDP while the program is running on the main processor (MP).

For MEMSY a new, so-called Extended Signature Integrity Checking (ESIC) method of control-flow checking using assigned signatures was developed. It is an extension of the signature integrity checking (SIC) method described in [16]. As in the original SIC method a preprocessor extracts the so-called control flow graph from the high level programming language source code [11], [18]. In this graph nodes represent branch-free program blocks (instruction sequences) and arcs correspond to control transfers (e.g. branch-type instructions as if-then-else or case statements). To each program node an unambiguous label (signature) is assigned. The graph is translated into a WDP program, reflecting the control structure of the main program. At all branches of the main program signature transfer statements are inserted into the source code in order to transfer node labels to the WDP during main program execution. These signatures identify the current main program location. Similarly, in the WDP program receive_signature statements are inserted at the corresponding positions. Both programs are compiled into machine codes of the target and watchdog processor, respectively.

The MP and the WDP are started simultaneously and then they execute the main and the watchdog programs, respectively. The WDP monitors concurrently the correct execution order of the main program blocks by checking the received actual signature sequence. A sequence of signatures is accepted as correct, if it corresponds to an existing path in the program graph. In case of a conditional branch instruction, it is only checked, whether the target instruction belongs to the set of admissible target instructions.

We implemented the WDP as a finite deterministic push-down automaton. For this automaton a preprocessor extracts the program graph in a tabular form which defines for each current signature (state) the set of the allowed subsequent signatures.

In order to overcome problems related to subroutines, the preprocessor generates a separate control flow graph for each subroutine and a corresponding, separate automaton table. The start and the end node of every subroutine are labelled by special signatures, viz. "start of subroutine" (SOP) and "end of subroutine" (EOP), respectively. This signatures contain a field uniquely identifying the subroutine.

If a SOP-signature belonging to an existing subroutine entry point is received, the current state is pushed onto the WDP private stack and the new state of the WDP is the initial state (start node) of the subroutine, i.e. the WDP switches over to the automaton table of the subroutine.

If the WDP receives an EOP-signature it checks, whether the signature belongs to the current subroutine, and if so its next state is popped from the private stack, i.e. the WDP resumes checking the calling program.

2.3.3. WDP hardware and measurement results

To demonstrate the usefulness of ESIC for MEMSY a WDP prototype was built based on a

2.3.4. Simultaneous use of master-checker mode and watchdog processors

As described before, the MC-mode assures high fault coverage for the computing core, but leaves the system unprotected against faults in other parts. A WDP provides a moderate protection against faults in the overall system resulting in control flow errors, but can not detect pure data errors and data related control flow errors such as the selection of a wrong branch in an if-then-else statement.

If both methods are applied simultaneously, the signatures are sent by the main processors to the WDP and are checked by the checker. Therefore, even data related control flow errors can be detected, if they originate from faults in the computing core. The main results of the simultaneous use of the MC-mode and the watchdog processor are summarized in Table 2. Shaded parts denote the method dominating the fault coverage.

<table>
<tr><th colspan="2" rowspan="2">Fault location</th><th colspan="3">Fault coverage in %</th></tr>
<tr><th>Master-checker only</th><th>Watchdog only</th><th>Simultaneous use</th></tr>
<tr><td rowspan="2">CPU-internal</td><td>control</td><td rowspan="5">~ 100</td><td>~ 80</td><td rowspan="5">~ 100</td></tr>
<tr><td>data</td><td>0</td></tr>
<tr><td colspan="2">Code cache</td><td rowspan="2">50-80</td></tr>
<tr><td rowspan="2">Data cache</td><td>control</td></tr>
<tr><td>data</td><td>0</td></tr>
<tr><td rowspan="2">External components</td><td>control</td><td rowspan="2">0</td><td>50-80</td><td>50-80</td></tr>
<tr><td>data</td><td>0</td><td>0</td></tr>
</table>

Table 1 Fault coverage

As it is shown in this table, the only important uncovered errors are external data errors. However, such errors can easily be detected if the main memory is protected by a proper error-detecting code.

2.4. Interconnection Network

Based on the MEMSY topolgy 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-level has access to the communication memories of its four assigned A-level-nodes.

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 units supports the virtual implementation of the described topology and 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 one level.

Each coupling unit is a blocking, multistage, dynamic network with fixed size providing logically complete interconnections 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.

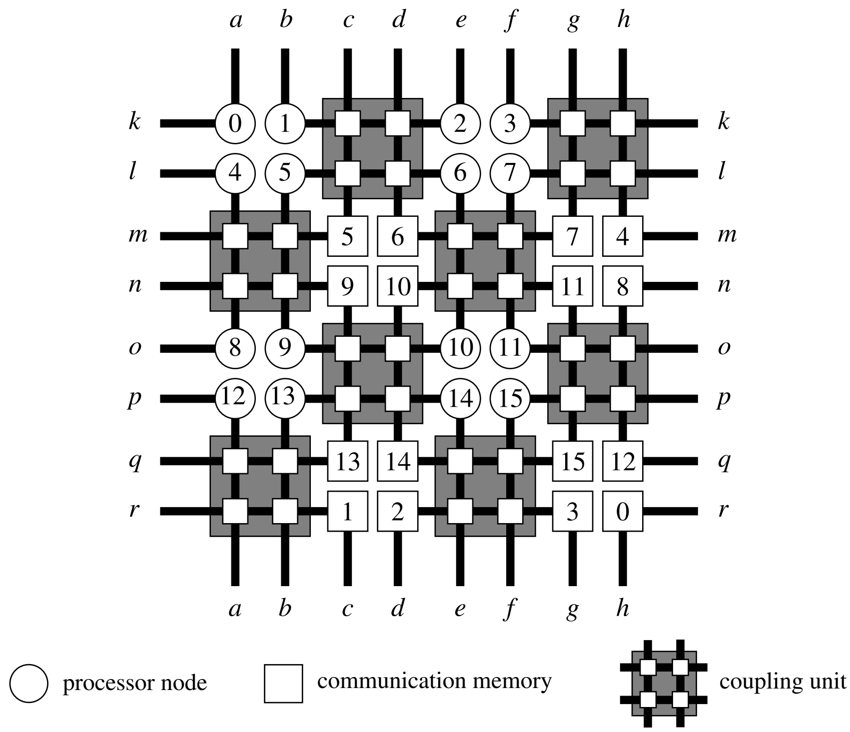

Fig. 3 Composition of processor nodes, communication memories and coupling units

The torus topology of a single MEMSY level is implemented by the arrangement of nodes, communication memories, and coupling units as shown in Fig. 3. 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-level to the four corresponding communication memories of the A-level are also implemented by using coupling units. These are connected to the third ports.

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.

2.4.1. Internal Structure of the Coupling Unit.

The hardware component used to implement a multiprocessor system, as described above, is shown in Fig. 4.

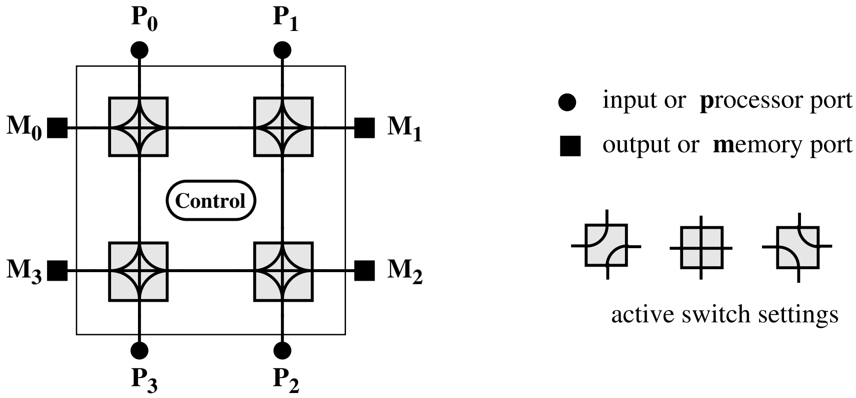

Fig. 4 Internal structure of a coupling unit

A coupling unit consists of the following subcomponents:

- 4 p-ports allowing the access to the coupling unit by nodes
- 4 m-ports providing the connection of communication memories
- 4 internal subpaths performing data transfer within the coupling unit
- 1 control unit which controls the dynamic interconnection between p-ports and m-ports
- 4 switching elements providing 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.

2.4.2. Fault Tolerance of the Interconnection Network

The coupling unit used as a building block of the MEMSY interconnection network [8] provides mechanisms at hardware level which support the efficient use of alternative communication paths in case of faults. The basis for this fault tolerance feature is the ring structure of the internal subpaths. Alternative communication paths consist of disjoint sets of internal subpaths. Hence, the permanent failure of one internal subpath within a network unit can be tolerated. The result will be a reduced bandwidth but full interconnection is guaranteed (graceful degradation).

The failure of a single internal subpath is always tolerated. A simple example demonstrates the reaction to a faulty memory access. Consider the following scenario (cf. Fig. 5):
Processing node P_i is faultfree and performs an access to the communication memory CM_j. P_i observes the memory access as faulty. (For error detection, parity checking is used.) To mask

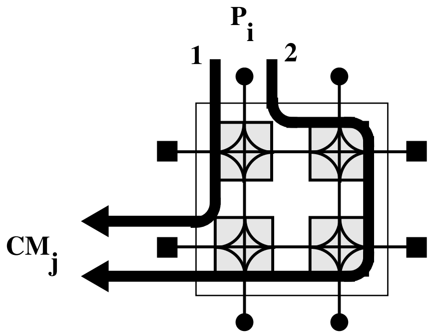

Fig. 5 Alternative communication paths from P_i to CM_j

a temporary fault the memory access is repeated. If the error still occurs the fault may reside either within the communication memory or within the interconnection network.

At this point it is not possible to decide which one has lead to the faulty memory access. To locate the fault the communication memory is accessed via different communication paths and the results are compared, Fig. 5.

At hardware level this possibility to localize faults is supported as follow: Each communication memory which is directly accessible by a processor is mapped into the processor's address space (Fig. 6) such that accesses to communication memory CM_j can be performed within the address subspaces AS_{jd}, AS_{j1} or AS_{j2}. The same offset address within any of these address subspaces will select the same memory cell but a different communication path within the network unit:

- AS_{jd}: dynamic selection of the communication path according to the current switch settings
- AS_{j1}: communication path 1
- AS_{j2}: communication path 2.

As long as no fault has been detected, dynamic selection of the communication path is used. This provides higher efficiency by avoiding conflicts within the network unit. In case of a fault,

address subspace		
	CM_{j+1}	(dynamic)
AS_{j2}	CM_j	(fixed 2)
AS_{j1}	CM_j	(fixed 1)
AS_{jd}	CM_j	(dynamic)
	CM_{j-1}	(fixed 2)

Fig. 6 Multiple mapping of communication memories

all accesses to communication memories must be carried out either within AS_{j1} or AS_{j2}. Which of these two address subspaces is selected is determined individually for every processing node.

Let us now consider stuck-at faults. A stuck-at fault of one switching element within the network unit may result in one of three situations depending on the static switch setting:

- Switch inactive: p-port and m-port are isolated (Fig. 7a)

- Switch connects p-port with m-port (Fig. 7b)
- Switch connects p-port and m-port with different internal subpaths (Fig. 7c, Fig. 7d)

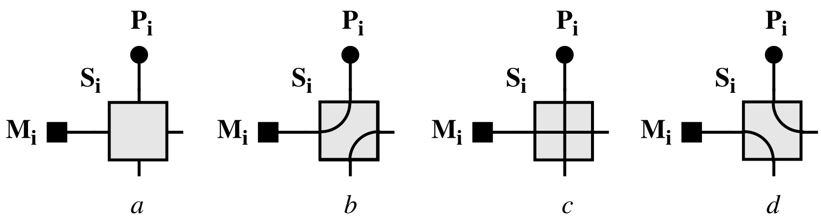

Fig. 7 Stuck-at faults of a single switching element of the network

Thus, a switch is either intact, inactive or stuck-at. The inactive state is equivalent to link failures. If switch S_i is inactive then processor P_i can still access its communication memory M_i by using its second port and a different network unit. Many single stuck-at faults can be tolerated within the network unit itself, Fig. 8 demonstrates this. Fig. 8a is the desired access pattern; Fig. 8b and Fig. 8c show how the stuck-at faults of switch S_0 are masked by the network unit. An analysis of the fault tolerance capacity of the interconnection network of MEMSY is given in [8].

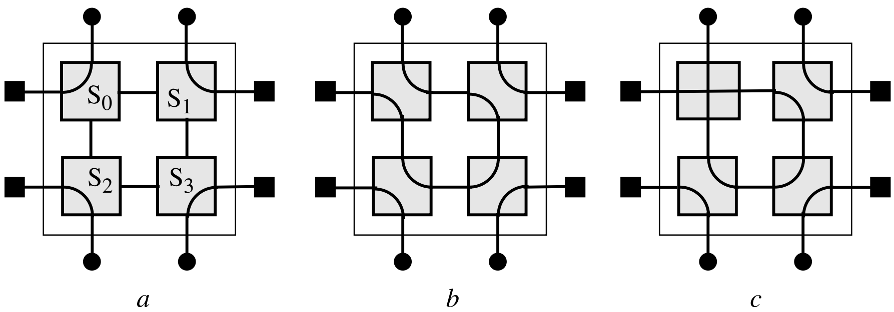

Fig. 8 Masked stuck-at faults

2.5. Performance of the MEMSY Prototype

The experimental system containing 20 processor nodes with a total of 80 processors offers the following performance data:

- 2000 MIPS (peak) approx. 1440 MIPS (permanent)
- 1000 MFLOPS (peak) approx. 800 MFLOPS (permanent)
- 10 MB physical cache
- 640 MB local memory (expandable to 3840 MB)
- 80 MB communication memory (expandable to 320 MB)
- 10 GB local disc (expandable at pleasure)
- 100 MB/sec I/O bandwith to the discs (expandable to 300MB/sec)
- 400 MB/sec to local memories
- 177 MB/sec to Communication memories

3. Programming Model

The programming model of MEMSY offers the application programmer direct access to the system structure. Unlike in many systems, where the programmer's concept of the system is different from the real structure of the hardware, the user of MEMSY should have a concept of the system which is very close to its real structure. This enables the programmer to write highly efficient programs which make the best use of the system.

The programming model is defined as a set of library calls which can be called from C and C++. MEMSY 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.

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 (i.e. controlled ordering of events). From these mechanisms the application programmer may pick the ones which are best suited for his particular problem (for details see [10], [11]):

Shared-Memory. The access to the shared-memory is based on the concept of 'segments', very much like the UNIX System V. A process which wants to share data with another process first has to create a shared-memory segment. After that, other processes may map the same segment into their address space by the means of an 'attach' operation.

Messages. This mechanism allows the programmer to send short (2 word) messages to another processor; they are mainly used for coordination and not especially optimized for high-volume data transfer.

Semaphores provide a simple method for global coordination; they reside on the node on which they have been created, but can be accessed uniformly throughout the whole system.

Spinlocks can be used for the coordination of short critical sections; they reside as two-valued variables (busy, available) in shared-memory segments.

Transport. This mechanism was designed for high volume and fast data transfer between any two nodes in the system. Contrary to 'message' it allows the transfer to more distant nodes.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 processor node has a local data storage area. There is one global data storage area common to all the 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'.

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. Operating System

The operating system of MEMSY - MEMSOS - is based on the UNIX SYSTEM V/88 Release 3 of Motorola. Instead of the decision to design and implement a completely new operating system we have decided to use an existing system and adapt it to the new hardware. Unix supplies a good development environment and many useful tools. The multitasking / multiuser feature of Unix is included with no additional effort.

Each processor node has its local operating system 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.

The original operating system is not able to deal with distributed memory such as our communication memory. Therefore certain extensions and additions have been made to it. 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. For further details of the support of distributed user programs by the so-called *application concept*, the interrupt and message-passing mechanisms see [10], [11].

The 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.

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.

5. Fault Treatment

The rollback-recovery scheme for memory-coupled multiprocessors as realized for MEMSY is based on the notion of distributed snapshots [2], [10]. The state of an application program is defined by the state of every participating process plus the states of shared memory segments. Other approaches, such as conversations [16], [17] or message logging schemes [9] are not considered suitable, the latter because logging is impossible, as shared-memory communication does not necessarily involve a message passing programming model.

Furthermore, we consider numerical applications as the primary use of MEMSY. These applications are characterized by high communication loads. Most numerical applications are basically iteration loops, offering suitable spots for global checkpointing. Thus, to optimize checkpointing, we assume the programmer specifies the type of data to be stored and the exact places within the execution of a program, when checkpoints shall be taken. A fully transparent scheme would cost much more coordination overhead.

Our primary goal of an implementation of the rollback-recovery mechanism is achieving as high an efficiency as possible, that is, we want the mean runtime of a job on a fault-ridden, fault-tolerant system (T_{FF}) to be about as long as the runtime of the same job on a fault-free system with no fault tolerance mechanisms (T_0). The efficiency can be defined as:

$$E = \frac{T_0}{T_{FF}}$$

Our second goal is a high trustworthiness of the results of a long-running computation. The means to reach this goal are provided by the error detection mechanisms described earlier in this paper. We expect 99% of all faults to be detected.

To reach the goal of high efficiency two mechanisms are employed:

- two-level checkpointing scheme with ‘hard’ and ‘soft’ checkpoints
- hardware-assisted, non-blocking coordination of global checkpointing

The two-level checkpointing scheme is based on the assumption, that there are different media for storing checkpoints available: fast, unreliable storage and comparatively slow, stable storage. Fast storage can be the local memory (main and background storage) of a processing node, while stable storage can be replicated filesystems on a remote server or stable RAM (see section 3.2). Error recovery from the local, unreliable checkpoint storage, would not always succeed, but may fail with a certain probability, whereas recovery from stable storage is supposed to succeed always.

Another aspect of numerical applications is the large size of data structures. We therefore assume, that checkpoints occupy a considerable amount of space in the storage medium, imposing an upper limit on the number of checkpoints kept in storage. Thus, global coordination is necessary, to determine at which times checkpoints are obsoleted and may be discarded.

Global checkpointing requires all processes of an application to record their state atomically, even if errors during checkpointing occur. A decentralized two-phase commit protocol is employed to ensure a new checkpoint has been taken before outdated checkpoints are discarded. As decentralized two-phase commit protocols require efficient broadcast algorithms, special support hardware reducing the number of broadcasts is presently under development.

5.1 Fault Classes

We divide faults into two classes: ‘soft’ faults and ‘hard’ faults. Soft faults allow global recovery from state information kept locally, while hard faults require rollback to checkpoints from stable storage. Hard faults would include hardware failures and node crashes, as node crashes cause the loss of local memory data. The two-level checkpointing scheme is based on the assumption that most errors fall into the soft category. This assumption is based on several observations.

First, once detected, hardware errors cause exceptions, caught by the operating system of a node. The operating system then determines the cause of the exception. If the exception occurred in user mode, the application program is signalled, otherwise, in most cases, the node will crash. Yet it is safe to assume that the amount of resources, such as cpu time, spent on the execution of the operating system in kernel mode is far less than the amount spent on the execution of the application. Most transient hardware errors, therefore, will fall into the soft category.

Permanent faults, such as node failures, fall into the category of hard errors. Reconfiguration and subsequent recovery is not possible from the local checkpoints, as the state information in local storage of the defective node cannot be accessed from other nodes. Studies in [18] show the frequency of permanent hardware errors about ten times less than the frequency of transient errors. Even though operating system software errors and their manifestation are an unknown quantity, but it is safe to expect that at least 75% of all errors are soft errors.

5.2 Two-level Rollback Scheme

There are several advantages to a two-level rollback algorithm. First, soft checkpoints are created comparatively quickly, because of the use of fast storage media. Besides soft checkpoints can be written in a ‘careless’ fashion, as there is no need to guarantee sure recovery. Fast checkpointing reduces the length of the optimal checkpointing interval, thus reducing the loss of job progress in case of errors. Shorter intervals also reduce the probability of the decay of checkpoint data, as checkpoints are kept for a shorter time. As recovery with a soft checkpoint may fail, efficiency is reduced resulting in increased job runtime.

As a remedy, hard checkpoints are created. In case of successful ‘soft’ rollback the soft checkpoint has been proven to be intact, thus, whenever rollback to a soft checkpoint succeeds, this checkpoint is made a hard checkpoint, by moving it to stable storage. As ‘hard’ rollback is considered to be always successful, even for very long job runtimes the efficiency drops asymptotically to a comparatively high value. On the other hand, hard checkpoints are saved seldom, one hard checkpoint per mean-time-to-failure (MTTF) on average. Thus, the impact of slow access to stable storage on overall runtime is kept low.

Fig. 6 shows the efficiency of different checkpointing schemes, versus the runtime of a job, expressed in multiples of a system’s MTTF. The efficiency is defined as above. Value ‘1’ denotes the efficiency of a fault-free no fault-tolerance system. The first curve (‘no fault-tolerance’) should better be labelled as ‘almost no fault tolerance’, as some means of fault detection are employed: The job is rerun, until the same result has been computed at least twice. The second example, denoted as ‘restart’ examines the case in which errors are detected (at no time cost) and where, in case of an error, the job is restarted from the beginning. The third example (‘soft checkpointing only’) shows the efficiency of a checkpointing algorithm, where soft checkpoints are taken and rollback to a soft checkpoint succeeds with a probability P_s=7/8. Finally, the last example (‘hard & soft checkpoints’) shows the efficiency of the proposed checkpointing scheme.

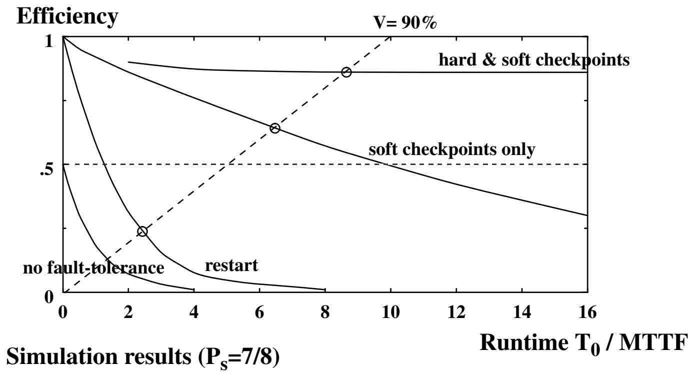

Fig. 9 Efficiency of several checkpointing schemes

We define the trustworthiness of a computation result as the probability that no error during the job runtime has gone undetected. The dotted line shows the boundary, after which this probability falls below the 90% mark, assuming a coverage of 99%. The trustworthiness of a job with a runtime equal to 8 times the MTTF is higher than 90% if hard and soft checkpointing is employed, and less than 90% if only soft checkpointing is used. Interesting consequences are that any long-running computation requires a coverage of error detection mechanisms significantly higher than 99%, and that efficient recovery mechanisms allow longer job runtimes. In our example the line of 90% trustworthiness is intersected at a higher value of job runtime for mechanisms with higher efficiency.

5.3 Parallelized Checkpoint Coordination

As progress of the application program does not depend upon the result of checkpointing, the two-phase commit protocol can run in parallel with the application, avoiding busy wait for coordination messages. Furthermore, global synchronisation at the end of each checkpointing interval is avoided. Otherwise, every process would be forced to wait for the slowest participant, thus penalizing overall job runtime, if the participants have different runtimes in each checkpoint interval.

With parallel checkpoint coordination the progress of processes is allowed to drift apart, limited by the amount of storage assigned for checkpoints. In the minimal configuration at least two checkpoint buffers have to be provided. One buffer is needed for the 'active' checkpoint. In case of faults the application rolls back to this checkpoint. The coordination protocol guarantees, that the active checkpoint has been established globally. A second buffer is needed for the 'tentative' checkpoint, a checkpoint which exists locally, but about which it is not known, whether every other process reaches the same point of progress. The tentative checkpoint might be committed eventually and become the new active checkpoint, or it might be aborted and removed. Even with only these two buffers different processes can be allowed to drift one checkpoint interval apart, as Fig. 7 shows.

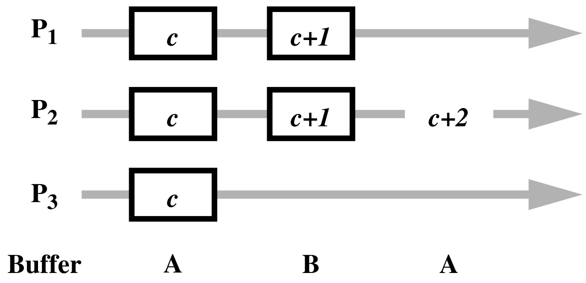

Fig. 10 Usage of two checkpoint buffers

Process P_2 is the farthest ahead. P_2 tries to establish checkpoint number c+2, but is not allowed to, as buffer A is still occupied by the active checkpoint with number c. It has to wait for P_3 to establish checkpoint c+1 as well. After c+1 is committed and thus made the active checkpoint, buffer A is freed and P_2 is allowed to proceed.

5.4 Hardware-assisted Broadcast

Two-phase commit with a decentralized communication structure requires all participants to broadcast their local state. Afterwards, every participant decides independently what to do with the tentative checkpoint, according to the information received locally. In order to make sure that every participant reaches the same decision, the broadcast algorithm has to be reliable and has to guarantee the same ordering of messages. For MEMSY yet another path has been chosen: a special broadcast hardware, based on an optical bus, is provided to allow simultaneous broadcasts: while every participant is sending its local state, it receives the global state at the same time. In a way rather similar to a wired-OR network, any single participant can force the whole application program to abort a tentative checkpoint.

6. Conclusions

Acknowledgements

References

[1] Banâtre, M.; Muller, G.; Rochat, B.; Sanchez, P.: Design Decisions for the FTM: A General Purpose Fault Tolerant Machine, Proc. 21th FTCS, pp. 71-78, 1991

[2] Baron, R.J.; Higbie, L.: Computer Architecture. Addison-Wesley, 1992

[3] Chandy, K. M.; Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, ACM T.o.C.S., vol. 3, no. 1, pp. 63-75, 1985

[4] Cristian, F.: Understanding Fault Tolerant Distributed Systems, Com. ACM vol. 34 (1991), pp. 56-78

[5] M. Dal Cin, W. Hohl, E. Michel, A. Pataricza: Error Detection 1993

[6] Dal Cin, M.: New Trends in Parallel and Reliable Computing: Massive Parallelism and Fault Tolerance. Invited paper, Proc. µP'92, 7th Symposium on Microcomputer and Microprocessor Appl., Budapest, April 1992, pp. 1-10

[7] Dal Cin, M.; Grygier, A.; Hessenauer, H.; Hildebrand, U.; Hönig, J.; Hohl, W.; Michel, E.; Pataricza, A.: "Fault Tolerance in Distributed Shared Memory Multiprocessors", in: A. Bode, M. Dal Cin (eds.), Parallel Computer Architectures - Theory, Hardware, Software, Applications, Springer Lecture Notes in Computer Sciences 732, 1993, pp. 31-48

[8] Grand Challenges: High Performance Computing and Communications. The Fiscal Year 1992 U.S. Research and Development Program. Report by the Committee on Physical, Mathematical, and Engineering Sciences, NSF Washington 1992

[9] Hildebrand, U.: A Fault Tolerant Interconnection Network for Memory-Coupled Multiprocessor Systems, In: Dal Cin, M.; Hohl, W.(eds.): Proc. 5th Int. Conf. Fault Tolerant Computing Systems, Informatik-Fachberichte 283, pp. 360-371, Springer 1991

[10] Hofmann, F.: Das Querschnittsprojekt "MEMSY", Ein Modulares Erweiterbares Multiprozessorsystem, Proc. Euro-Arch '93, pp. 567-577, Informatik-aktuell, Springer 1993

[11] Hofmann, F.; M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, C.-U. Linster, T. Thiel, S. Turowski: MEMSY - A Modular Expandable Multiprocessor System, in A. Bode, M. Dal Cin (eds), Parallel Computer Architectures pp.15-30, Sringer LNCS 732, 1993

[12] Hohl, W.; Michel, E.; Pataricza, A.: "Hardware Support for Error Detection in Multiprocessor Systems - A Case Study", Microprocessors and Microsystems, Vol.17, No. 4, 1993, 201-206

[13] Kai Li; Naughton, J. F.; Plank, J. S.: Checkpointing Multicomputer Applications, Proc. 10th Symposium on Reliable Distributed Systems, pp. 2-12, 1991

[14] Koo, R.; Toueg, S.: Checkpointing and Rollback-Recovery for Distributed Systems, IEEE T.o.S.E., pp. 23-31, Jan. 1987

[15] Lampson, B. W.: The Stable System, in Lampson, B. W.; Paul, M.; Siegert H. J. (ed): Distributed Systems: Architecture and Implementation, LNCS 105, pp. 254-256, 1988

[16] Leveugle, R.; Michel, T.; Saucier, G.: Design of Microprocessors with Built-In On-Line Test, Proc. 20th FTCS, pp. 450-456, 1990

[17] Lu, D. J.: Watchdog Processors and Structural Integrity Checking, IEEE T.o.C., Vol. 31. No.7, 681-685, 1982

[18] Mahmood, A; McCluskey, E. J.: Concurrent Error Detection Using Watchdog Processors - A Survey, IEEE, T.o.C., Vol. 37. No. 2, pp. 160-174, 1988

[19] Michel, E.; Hohl, W.: Concurrent Error Detection Using Watchdog Processors in the Multiprocessor System MEMSY, Proc. 5th Int. Conf. Fault-Tolerant Computing Systems, Nürnberg, Informatik Fachberichte 283, pp. 54-64, Springer, September 1991

[20] Pataricza, A.; Majzik, I.; Hohl, W.; Hönig, J.: "Watchdog Processors in Parallel Systems", to be published in the Euromicro Journal

[21] Pataricza, A.; Majzik, I.; Dal Cin. M.; Hohl, W.; Hönig, J.: "Fast Watchdog Processors for Multiprocessor Systems", submitted to the FTCS-24, Austin, 1994

[22] K. Rusnock, *The Multiprocessor M88000 RISC Product*; Motorola Microcomputer Division, Tempe, AZ 85282, 1991

[23] Russell, D. L.; Tiedeman, M. J.: Multiprocess Recovery Using Conversations, Proc. 9th FTCS, pp. 106-109, 1979

[24] Shrivastava, S.; Mancini, L.; Randell, B.: On The Duality Of Fault Tolerant System Structures. In: J. Nehmer (ed.), Experiences With Distributed Systems, Proc. Int. WS. Kaiserslautern 1987, pp. 10-37, Springer LNCS 309, 1988

[25] Siewiorek, D. P.: Faults And Their Manifestation, Springer LNCS 448, pp. 244-261, 1987