North-Holland
Microprocessing and Microprogramming 20 (1987) 301–308

ATTEMPTO: An Experimental Fault-Tolerant Multiprocessor System

M. Dal Cin, R. Brause, J. Lutz
J.W. Goethe University of Frankfurt, West Germany
and
E. Dilger, Th. Risse
Eberhard-Karls University of Tübingen, Wilhelmstr. 7, 7400 Tübingen, West Germany

This paper describes the overall hardware and software architecture of a fully decentralized, fault-tolerant system. It provides a single-user multi-tasking computing environment. Currently, the system is intended for use as a test-bed for fault-tolerant computing.

Keywords: Multiprocessor, Operating system, Fault tolerance, Fault diagnosis.

1. Introduction

With the rapid decline in the cost of computer hardware it is now feasible to dedicate a multi-processor system to a single user. Consequently, we felt that it makes sense to exploit the advantages of multiple resources provided by a multi-microprocessor system and to develop a single-user fault-tolerant computing environment. In addition, we felt that conventional architectures of fault-tolerant systems have some serious disadvantages:

- Fault-tolerance mechanisms must be explicitly known and used by the user in his programs. This implies special program changes and impedes third-party software.
- The architectural solution is often very specialized, not modular and, therefore, not portable.
- The hardware and software used do not conform to industrial standards.
- Fault-tolerance covers only part of the whole system.
- The system is simply too expensive compared with non-fault-tolerant versions.

In order to overcome this situation we adhered to the following design goals.

Our system has been named ATTEMPTO [1] (A TESTable Experimental MULTIProcessor with fault-TOlerance) and is intended to serve the research team as a test-bed for fault-tolerance mechanisms. The prevailing design goals are:

- The user himself should be able to decide for each application job to what extent it should run in a fault-tolerant environment.
- The mechanisms implementing fault-tolerance should be transparent to the user who sees the system as a multitasking monoprocessor system. Consequently all binary non-fault-tolerant programs must run without changes also in a fault-tolerant mode.
- All fault tolerance mechanisms – such as fault-diagnosis, voting or reconfiguration – should be fully decentralized in order that the systems survive the breakdown of a single component.
- The system is to be built from conventional hardware parts. Hence most of its fault tolerance is to be implemented in software.
- The fault tolerance mechanisms must be modular and hardware-independent, thus allowing reconfiguration (for experiments on fault-tolerance) by adding or exchanging software and hardware components.
- Software necessary for fault tolerance mechanisms must be portable. It must be modular, well-structured and written in a high-level language.
- The operating system and the utilities must conform with standard systems.
- Fault tolerance must cover the entire system (excluding input and output lines).

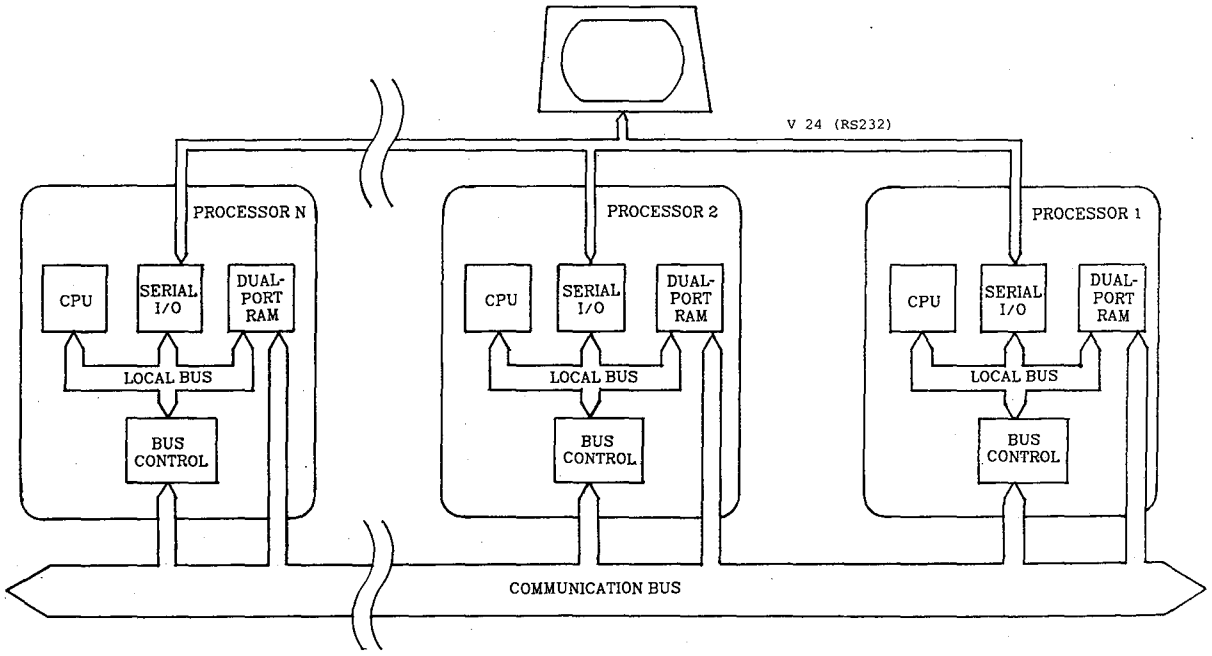

Fig. 1. Hardware

These requirements led primarily to the development of a modular, hierarchically structured operating system layer [2] which provides the fault tolerance services of ATTEMPTO.

After a brief overview of the system (Section 2) we will present the overall structure of the operating system layer (Section 3) and explain our concept of fault treatment (Section 4).

2. System Overview

2.1 Hardware

Single-board computers with dual port RAM were chosen as processing nodes (Fig. 1). Communication between these nodes is provided by a multi-master-bus. Our approach is, however, also imple-

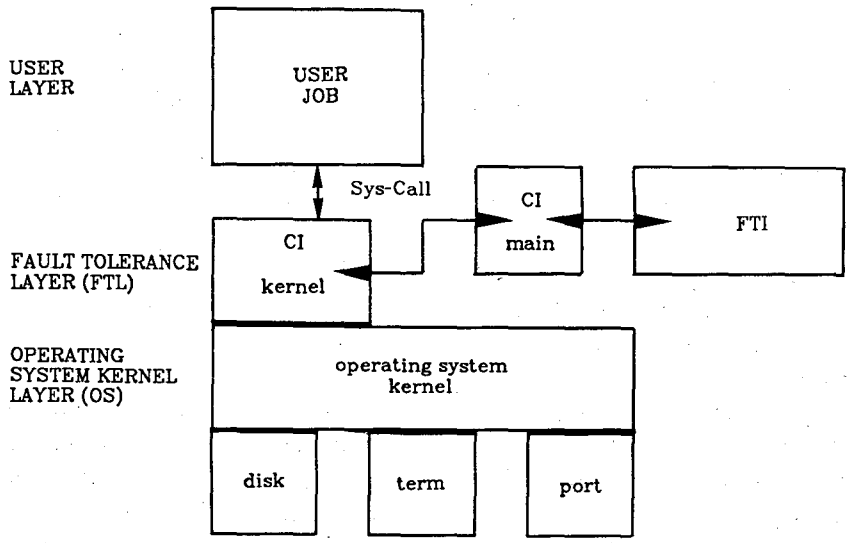

Fig. 2. ATOS. CI: Communication Instance; FTI: Fault Tolerance Instance.

mentable on a multiple bus system in order to enhance the system's fault tolerance with respect to bus errors. To ensure that none of the processing units damages the user's input data, the user input is directly available to all units (by connecting the user terminal to the serial i/o-port of each board). Via the dual-port RAM's a unique logical communication link is established between each pair of processing nodes. (The memory ports are used WRITE-ONLY on global and READ-ONLY on local addresses. The global base addresses of the memory ports are selected from an EC-Code, to hinder addressing of wrong ports by bit faults with memory, bus lines or bus arbiter as possible sources, cf. Fig. 3.)

For each processing unit there is one interrupt line on the communication bus. The sender of a message broadcasts its message over the communication bus to all concerned units, including itself. After transferring the data, it activates the interrupt line dedicated to it. This triggers the read of the message by all units providing an asynchronous, atomic transmission of messages; cf. Fig. 4. The temporal order in which incoming messages are accepted is the same for all processing units. It may, however, be different from the temporal order of their individual arrivals. A message transfer protocol specified in [3] establishes the base for this synchronisation of the processing units.

2.2 Operating System

ATOS is the node operating system of ATTEMPTO. It is comprised of two parts: The OS-Kernel and the Fault-Tolerance Layer (FTL) which is responsible for implementing the fault tolerance. This layer is transparent to the user and is programmed in Modula-2 [11]. Its location and connection to the operating system is shown in Fig. 2.

2.3 Job Management

The binding of application jobs to processing units

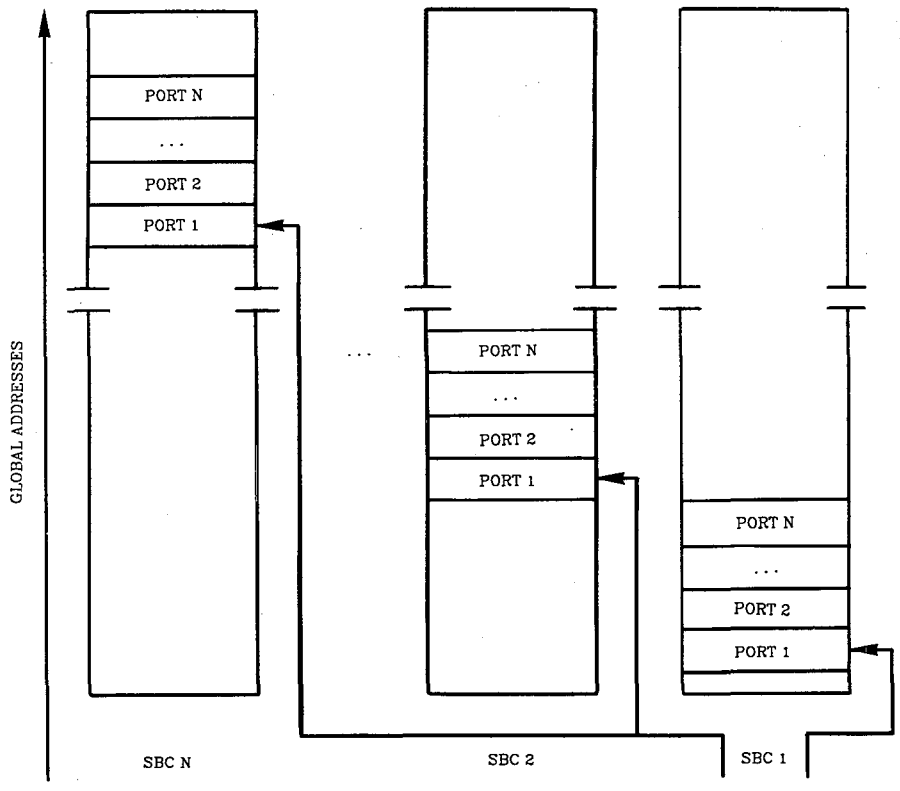

Fig. 3. Address space of message ports.

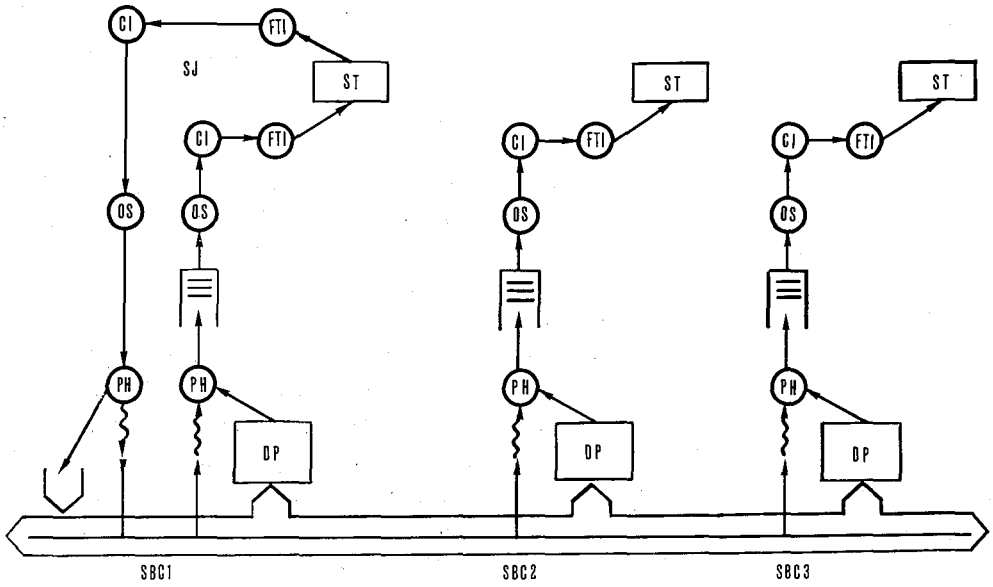

Fig. 4. Start-job (SJ) request by SBC1. ST: system table; OS: operating system; PH: port handler; DP: dual port memory: waved arrow: interrupt.

is transparent to the user. It is based on the principle of job attraction [10] (implemented in software) in order to avoid the need for centralized scheduling and dispatching. Each node maintains its own system tables, which it updates upon receiving messages from other nodes. An (idle) unit applies for the next job by sending a start request to all units including itself. Upon receiving such a request, it marks its system table entry corresponding to the job and responds to the requester. Requests for active jobs (i.e. jobs already being executed by $t+3$ units, see Section 4.2) are ignored; cf. Fig. 4.

3. The Fault-Tolerance-Layer

3.1 Structure

The FTL of ATOS itself is divided into several functional sublayers [2] (Fig. 5), viz:
- the Fault-Tolerance Instance FTI
- the communication support layer
- the service layer and
- the system layer.

The sublayers 3 and 4 consist of collections of specific Modula-2 modules. Each module comprises a data structure (e.g. Job Control Buffer, Data Input Buffer, Signature Array Buffer, etc.) and an active unit that maintains the data structure. Active units are referred to as Module-clerks and are Modula-2 processes. Clerks communicate by exchanging messages. The second pair of sublayers is composed out of a set of information concealing modules with strictly procedural interfaces.

Hence, the architecture of the higher part of FTL, is based on the message oriented model of Lauer and Needham [12] and that of the lower parts based on the procedure oriented model. From the viewpoint of the OS-kernel, the FTL is just another user process (with higher priority) that shares its proces-

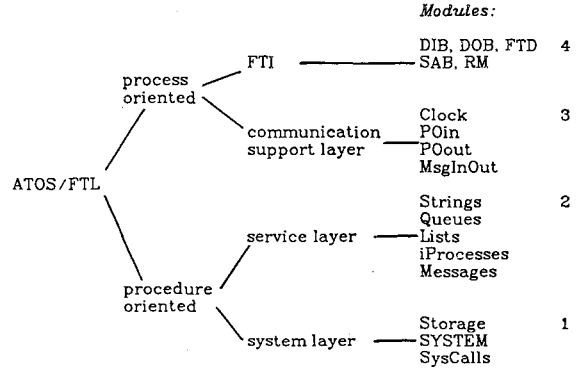

Fig. 5. Module hierarchy of the Fault Tolerance Layer.

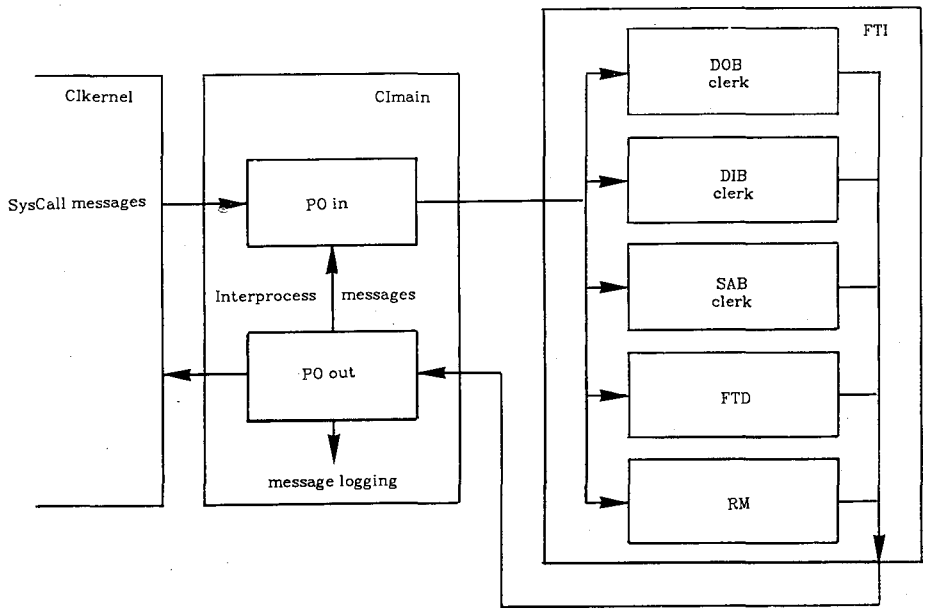

Fig. 6. Message exchange in the Fault Tolerance Layer.

sor time among several Modula-2 processes.

In ATTEMPTO we must distinguish between three levels of communication:

- communication between clerks (Modula-2 processes)
- inter-process-communication (UNIX processes)
- inter-processor-communication.

Communication between clerks on different SBCs implicitly uses all three levels.

The overall message exchange system of the Modula-2 processes within the UNIX process "FTL" is shown in Fig. 6.

The objective of the FTL is to provide fault tolerance if required by the user. The FTL also provides complete internal observability to the experimenter (but not to the ordinary user). Moreover, its modular and hierarchical structure and the fact that it is exclusively programmed in a higher level language allow us to substitute single modules by modules implementing different strategies for fault-treatment.

3.2 Sublayers

We now characterize very briefly the function of each sublayer. The FTI provides high level fault tolerance services. Its core is formed by the modules DIB, DOB, SAB and FTD. The DIB-Clerk manages the data typed in by the user (input-buffer) and prepares the user-job output data for fault-diagnosis. The DOB-Clerk manages the user job data output-buffer and forwards only the data which are diagnosed as being correct. The DOB-Clerk is authorized to do so by the SAB-Clerk. The SAB-Clerk handles all diagnosis tasks described in Section 4. To this end it maintains a so-called signature-array buffer. The FTD-Clerk (Fault-Tolerance Dispatcher) manages a Job Control Buffer and implements the principle of job attraction.

The communication support layer (Communication Instance, CI) is responsible for correct communication between an application job and its FTI as well as between the FTI's of different processing nodes. CI contains a module called Post Office (PO) which constitutes the interface to the OS-kernel. In order to send a message to another node a clerk sends this message to its Post Office. The PO-Clerk completes the message with additional information (e.g. the node-id) and delivers it via the OS-Kernel to the communication port handler. The PO-Clerk forwards also all incoming messages to the receiver clerks of the FTI.

The service layer provides services necessary for Modula-2-process management, buffer manage-

ment, resource management, etc. The system level provides services for storage management, messages, mailboxes, context switching and system calls.

It is worth mentioning that, although our technique is not specific to any particular implementation of the OS-kernel (it is only essential that the kernel is able to distinguish fault tolerance requests and local system calls), our prototype is intended to run under local UNIX-kernels. Roughly speaking, system calls are diverted to the FTL if fault tolerance requires this. The decision is made by the kernel routine CI-kernel which gains control again as soon as the fault tolerance service has been delivered by the FTL. This technique offers several advantages:

- Every runnable code can be executed fault-tolerantly without modifications in response to the user's wishes (cf. Section 4).
- Changes of the kernel that become necessary remain local and controllable since there is only one entry point into the kernel.
- The entry to the FTL is protected just as entries to the OS-kernel are.
- The kernel routine, CI-kernel, can easily be attached to any operating system kernel (pseudo device).
- The method is more or less machine independent.

During development we emulated the system on a minicomputer. Currently the emulation is being upgraded in order to serve as a test-bed for other fault-tolerance purposes and an implementation using single-board-computers with Motorola 680XX and UNIX is under development.

4. Fault Diagnosis and Treatment

With regard to fault-treatment we adhere to an end-to-end strategy [4]. That is, the algorithms which implement fault tolerance are triggered not before the user-job charges the OS with a WRITE operation. Copies of an application (user) job are executed asynchroneously in parallel by several processing nodes and fault diagnosis is based on the so-called job-result comparison approach [5]. Idle nodes perform self-test routines. This approach is conceptually simple and independent of the hardware structure and of failure types. Several distributed diagnosis protocols for job-result comparison have been investigated and verified by Time-Petri-Net analysis [6].

Nodes executing copies of an application job form a single virtual processing node. The size of the virtual node is related to the so called degree t of fault tolerance (for the user's job) defined as the maximal number of node breakdowns which can be tolerated (in ATTEMPTO we have t = size of virtual mode - 3 for size > 3, $t = 1$ for size $= 3$, and $t = 0$ else). The degree of fault tolerance can be specified by the user at program start. E.g. typing in "MYPROGRAM#2#" means that "MYPROGRAM" should run with fault tolerance degree 2.

4.1 ATTEMPTO*'s Diagnostic Model*

The classical Preparata-Metze-Chien-Model [7] (PMC-Model) served us as the basis for the study of diagnostic methods that may be suitable for ATTEMPTO. This model is based on the idea that a system can be partitioned into subunits which test each other. In this case, a test consists of the transmission of a stimulus and of observing the reaction from this stimulus. It is implicitly assumed, that these tests are complete, i.e. that faulty units always show wrong reactions to test stimuli. However, due to the predetermined test direction and the assumption that tests must be complete, this model was abandoned for the use in ATTEMPTO. The following considerations played a role in our decision: It is not necessary for the users of a fault-tolerant system that the system is always functioning correctly. Important for the user is that the answers he receives from the system are correct. Hence, not all errors of the system must be treated immediately, rather, just those which make themselves apparent in contact with the environment (end-to-end-strategy).

Therefore, a new diagnostic model based on comparison tests was developed. It forms the basis of the diagnostic procedure used in ATTEMPTO. Comparison tests are employed before any output (WRITE-) operation.

In the PMC-model it also is implicitly assumed that a reliable subunit exists – the so-called "golden unit" – which decides on the basis of the test results which of the subunits are faulty. This assumption rarely applies in real systems. Therefore, we substituted the central diagnosis model of [7] by a decentralized one.

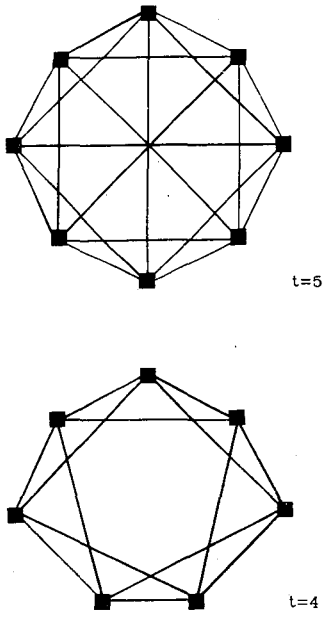

Fig. 7. Optimal diagnosis graphs. Connected squares: comparison pair.

4.2 Decentralized Diagnosis

The diagnosis of ATTEMPTO begins with the selection of pairs of subunits for comparison. Subunits perform the following steps:

Two units which are specified by the diagnosis algorithm compute the same algorithm (user or test program). The respective results are subsequently exchanged and compared. If both results are identical, then both units are assumed to be correct. Rather than to compare all subunits the smallest possible number of pairs is considered. To this end, all possible test assignments have been modeled and analyzed by (undirected) diagnosis graphs similar to the graphs of the PMC model. For ATTEMPTO strictly t-diagnosable, t-optimal diagnosis graphs were chosen [8]. These graphs are the basis of the distributed diagnosis in ATTEMPTO and, for a maximum of t faulty units, are optimal regarding the number of comparisons. (Recall that t is given by the user).

Fig. 7 shows two t-optimal graphs with $t=4$, $N=7$, and $t=5$, $N=8$, respectively (N number of units executing identical jobs).

Every unit sends its results to its neighbors (in accordance with the choosen graph), receives its neighbors results and compares them with its own results. Since the diagnosis graphs are strictly t-optimal, there are at least two units which are neighbors and which can immediately identify themselves as fault-free provided that altogether not more than t units are faulty. All units with erroneous results are neighbors of at least one of these fault-free units. Consequently the fault-free units recognize all faulty ones. A unit with an erroneous result will not find a neighbor with the same result. Nevertheless it may consider itself fault-free.

In order to hinder a faulty unit from passing on its result to the user, further message exchange is necessary. Each unit requires a key (e.g. the initial address of the output routine) in order to output. This key must be sent to it by another unit. (More precisely, unit i asks unit j for the key. Then unit j returns a message in which the desired key is encoded such that unit i can find the key only if its result coincides with that of unit j. Hence, a faulty unit will not be able to find the key. Recall, that two units are assumed to be faultfree if they produce identical results.) The unit which receives the key first is allowed to output. This output is monitored and compared by the other units. In order to limit the bus traffic in ATTEMPTO, the results are compressed to a normed length before they are sent and compared. For data compression a software version of a linear feedback shift register is used as follows:

A data package of length k whose bits are interpreted as coefficients of a polynomial of degree k-1 is divided by a given polynomial of degree r ($r=16$ in our system) with at least two coefficients $\neq 0$. The remainder of this division is the signature. Two data packages which differ by one bit produce different signatures [9]. Therefore, all one-bit-faults are detectable. If we, furthermore, assume that all possible faults in a data package are equally possible, we obtain a very low probability P that correct and faulty data packages are not distinguishable by their signatures, viz:

$$P = \frac{2^{k-r}-1}{2^k-1} \approx 10^{-5}$$

As it can be seen, this probability becomes independent of k for large k. Therefore, it is reasonable to compare signatures of large blocks of output data rather than bits or words. This decreases the bus

traffic and substantiates our diagnosis assumption that no two faulty units compute the same signatures.

5. Conclusion

In ATTEMPTO the algorithms which implement fault tolerance are triggered, each time a user-job charges the operating system with a WRITE-operation. Consequently, faults of individual subunits are ignored as long as outputs are not produced. Faults are diagnosed and masked using comparison tests just before they become noticeable by a false output or even by a missing one.

Designing ATTEMPTO we confined ourselves to considering only those fault-tolerant concepts which we felt to be fundamental and which did not require extensive hardware modifications. We are, however, convinced that the proposed combination of asynchroneous fault-masking with distributed fault-diagnosis compares favorably with techniques [13] such as checkpointing and rollback.

Acknowledgement

The authors gratefully acknowledge the help from Dr. E. Ammann and F.H. Florian.

The work has been supported by the Deutsche Forschungsgemeinschaft under Contract DA 141.

References

[1] Ammann, E., Brause, R., Dal Cin, M., Dilger, E., Lutz, J., Risse, T.: ATTEMPTO A Fault-Tolerant Multiprocessor Workstation: Design and Concepts, *Proc. FTCS-13*, Milano, pp. 10–13 (1983).

[2] Risse, T., Brause, R., Dal Cin, M., Dilger, E., Lutz, J.: Entwurf und Struktur einer Betriebssystemschicht zur Implementierung von Fehlertoleranz, *Informatik-Fachberichte* 84, Springer, pp. 66–76 (1984).

[3] Brause, R., Ammann, E., Dal Cin, M., Dilger, E., Lutz, J.: Softwarekonzepte des fehlertoleranten Arbeitsplatzrechners ATTEMPTO, *Symp. German Chapter of ACM, Microcomputing II*, Teubner Stuttgart, pp. 328–341 (1963).

[4] Saltzer, J.H. et al.: End-to-end Arguments in System Design, *Int. Conf. Distributed Computing Systems*, Paris, pp. 509–512 (1981).

[5] Ammann, E., Dal Cin, M.: Efficient Algorithms for Comparison-Based Self-Diagnosis, *Proc. Self-Diagnosis and Fault Tolerance*; Dal Cin M., Dilger E. (Eds.): ATTEMPTO Verlag Tübingen, pp. 1–18 (1981).

[6] Dal Cin, M., Florian, F.H.: Analysis of a Fault-Tolerant Distributed Diagnosis Algorithm, *Proc. FTCS-15*, Ann Arbor, pp. 159–165 (1985).

[7] Preparata, F.P., Metze, G., Chien, R.T.: On the Connection Assigment of Diagnosable Systems, *IEEE Trans. Electron. Comp. EC-16*, pp. 848–854 (1967).

[8] Ammann, E.: Vergleichstestmodelle für selbstdiagnostizierbare Systeme, *Informatik Fachberichte* 54, Springer, pp. 74–87 (1982).

[9] Smith, J.E.: Measurements of the Effectiveness of Fault Signature Analysis, *IEEE Trans. Comp.* C-29, pp. 510–514 (1980).

[10] Katsuki, D. et al.: PLURIBUS – An Operational Fault-Tolerant Multiprocessor, *Proceedings IEEE* Vol. 66, pp. 1146–1159 (1978).

[11] Wirth, N.: *Programming in Modula-2*, Springer (1982).

[12] Lauer, H.C., Needham, R.M.: On the Duality of Operating System Structures, *Operating Systems Review* 13, pp. 3–19 (1979).

[13] Randell, B.: System Structure for Software Fault Tolerance, *IEEE Trans. on Softw. Eng.* SE-1, pp. 220–232 (1975).

Rüdiger Brause is an assistant professor at the University of Frankfurt. He received an M.S. in Physics in 1978 and the Ph.D. in 1983 at the University of Tübingen.

He is now teaching courses on fault-tolerant computing and applications in vision, speech recognition and artificial intelligence.

Mario Dal Cin is a full professor for computer science at the Department of Computer Science at the J.W. Goethe-University Frankfurt, where he teaches courses on Parallel Processing, Reliability and Fault-Tolerant Computing. He received his Ph.D. in Physics from the University of Munich in 1969. From 1969 to 1971 he was a postdoctoral fellow at the Center for Theoretical Studies at the University of Miami. From 1972 to 1985 he was with the University of Tübingen.

Elmar Dilger is an assistant professor at the University of Tübingen. He received an M.S. in Mathematics and a Ph.D. in 1973 and 1977, respectively. He teaches courses on Automata Theory, Algorithms, Programming Languages and Fault-Tolerant Computing.

Joachim Lutz is a computer science Ph.D. student at the J.W. Goethe University Frankfurt. He received an M.S. in Physics from the University of Tübingen in 1982. His research interests include multiprocessor systems, distributed operating systems and fault-tolerant computing.

Thomas Risse received his Ph.D. in 1982 from the University of Tübingen. From 1980 to 1984 he was a research assistant at the Institute for Information Science. Since 1985 he is with the T.J. Watson Research Center, Yorktown Hights.