ATTEMPTO: A FAULT-TOLERANT MULTIPROCESSOR WORKING STATION; DESIGN AND CONCEPTS

E. Ammann, R. Brause, M. Dal Cin,
E. Dilger, J. Lutz, T. Risse

Institute for Information Sciences
University of Tübingen, FRG

Abstract

This paper presents the overall hardware and software structure of a fault-tolerant multi-processor working station which is currently being implemented at the University of Tübingen. In particular, job management for fault-tolerance and efficient fault detection and -localization by distributed fault-diagnosis based on comparison tests are described.

1. Introduction

ATTEMPTO +) is an experimental, fault-tolerant multi-processor working station under development in Tübingen. Our primary objective is to design a high-performance parallel computing facility which offers the user the possibility to turn the system into a fault-tolerant one on occasions where high dependability is required. Of course, the user can achieve this only if he is willing to spend part of his resources for reliability purposes, thus reducing the throughput of his system. Therefore, our second design goal is to enable the user to choose an appropriate balance of trade-offs in terms of throughput and fault-tolerance with respect to his applications. Hence, we are not aiming at an ultra-reliable system. Instead of providing ultra-reliability, the system's fault-tolerance should primarily ease maintenance by shortening the maintenance costs and response time and by making unscheduled maintenance a rarity.

Since the system is to be built from conventional computer components, most of its fault-tolerance mechanisms have to be implemented in software /10/. Only minor hardware additions will be required (see Sec.3). Therefore, our third objective is to demonstrate that fault-tolerance for a multi-processor working station comprising off-the-shelf single-board computers (SBC) can be implemented with help of a high level programming language (like Ada or Modula-2) without impeding the system too much.

Copies of a user job are executed asynchroneously in parallel by several processors; we call them colleagues. In ATTEMPTO "asynchroneously" means that processors communicate via messages only, each processor handles its scheduling and dispatching tasks on its own and a processor may not delegate tasks to other processors. Thus, in spite of the hardware used (cf. Sec.3), one could say that the processors are loosely coupled. This is important to prevent error propagation.

+) ATTEMPTO : A Testable Experimental Multiprocessor System with Fault-Tolerance, motto of the founder of the University of Tübingen

2. The Users View of ATTEMPTO

The user views our system as a single-user, multi-tasking system - its realization as a multiprocessor system is hidden.

He can choose the amount of fault-tolerance provided by ATTEMPTO depending on the needs of applications /5,11/. Having decided on the appropriate tolerance degree t for his job, all he has to do is to communicate it via the terminal bus (Fig.1) to the machine. Syntactically, the tolerance degree is part of the job name (Typing, e.g., 'MYJOB(2)' indicates that transient or permanent faults in up to t=2 different SBCs should be tolerated during the execution of the job such that no incorrect output is produced). Of course, t may not exceed a certain limit given by the available hardware redundancy. The user will be informed by the system if it is impossible to fulfill his fault-tolerance requirements. The user may also choose t=-1 indicating that he is content with fault-detection only. (The default value of t is t=0). Moreover, he has the option to initiate system-wide diagnosis and reconfiguration whenever this becomes meaningful.

The user can pass to the system as many jobs as the system's capacity permits -- several jobs with low or only a few jobs with high tolerance degrees.

3. Hardware Concepts

To satisfy the requirement for message-coupled autonomous units, Intel iSBC86/12a boards with dual-port RAM were chosen as single board computers. Communication by memory-coupling is provided via the multi-master bussystem MULTIBUS with priority logic (Fig.1).

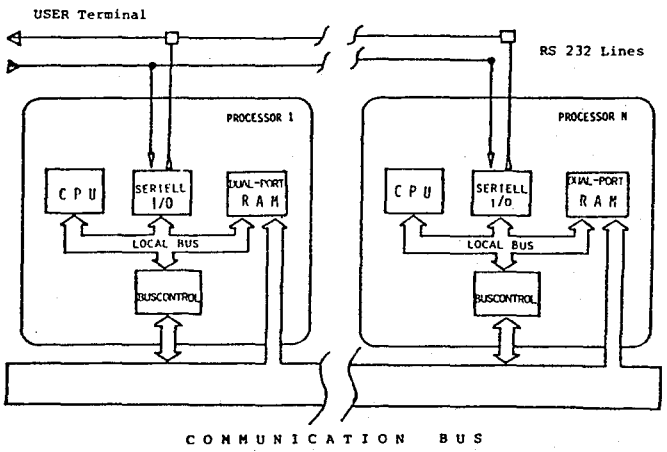


Fig. 1 Hardware configuration of the implementation

0731-3071/83/0000/0010$01.00 © 1983 IEEE

IEEE Proc. Int. Conf. Fault-Tolerant Comp. Syst. FTCS-13, Milano 1983

To ensure that no processor damages the user input data, the user input is directly available to all units by connecting the wires of the user terminal to the serial input of every SBC. Access to this "terminal bus" is managed via the corresponding resource semaphore in the system tables.

Fig. 2
Software layers

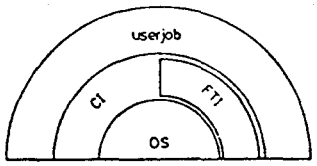
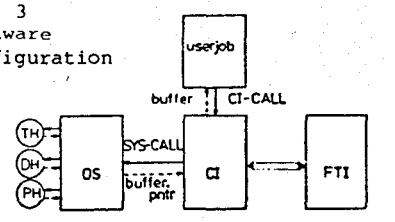


Fig. 3
Software configuration



TH = terminal handler
DH = disk handler
PH = port handler
solid lines = requests
dashed lines = responses
double line = messages

3.1 Inter-Processor Communication

On each SBC exists a memory region (dual-port RAM) which can be addressed globally via the communication bus, or directly on board. This region is used for communication as follows:

For each SBC of the system a subrange ("dedicated port") of this region is reserved for messages. For system synchronization purposes (cf. Sec.4.2) each SBC maintains a port for itself ("pseudo-port"). Hence, there is a unique communication link between each pair of SBCs which does not interfere with other communication links (disregarding the communication bus). The port region of the memory is used WRITE-ONLY on global and READ-ONLY on local addresses.

To hinder an addressing of wrong ports caused by bit faults (with memory, bus lines or bus arbiter as possible sources for faults) the global base addresses of the ports are chosen with pairwise Hamming distance of at least three.

3.2 Inter-Processor Synchronization

To avoid system crashes caused by an unique system table located in faulty memory each SBC has its own system table and updates it upon receiving messages from other SBCs. This avoids also system crashes caused by an access to a global system table via a faulty communication bus. To ensure consistent decentralized system tables it must be guaranteed that the time sequence of updates, and therefore, of incoming messages is identical on each SBC. We do not want to achieve this by LOCKing the communication bus. We choose rather the following logic protocol:

For every processor of the system there is an interrupt line on the communication bus. This line is activated after a message is transmitted to all receiver ports. Only after the interrupt event is the message read by the receiver's port handler (Sec.4.2). This allows an asynchroneous transmission of messages. The temporal order in which incoming messages are enregistered is not determined by their inidividual physical arrival times but by the sequence of the corresponding interrupts.

Thus, it is no longer important in which sequence messages are transmitted by the communication bus; they might even be interleaved by the physical bus protocol. No systemwide clock is needed.
Communication is only provided for fault-tolerance purposes and resource management.

4. System Software

The operating system of ATTEMPTO consists of identical, autonomous, local operating systems (ATOS=ATTEMPTOs local OS) - one on each SBC. ATOS's kernel is a single processor multitasking operating system (OS). On top of it there is a layer of system functions which provide the fault-tolerance properties.

4.1 The Intermediate Layer

ATOS consists of different levels (cf. Fig.2). Each one may be considered as a virtual machine. The intermediate layer consists of the Fault-Tolerance- (FTI) and the Communication- Instance (CI). For normal requests this layer is transparent. Its modules are programmed in Modula-2 which was chosen because of its powerful system design tools.

4.2 Communication instance

CI manages communication in ATTEMPTO System (Fig.3). A user job demands system service (e.g. input/output) by a CI-Call. The CI examines the request and protects the system against errors by checking syntax and admissibility of a request. The FTI gets requests from the CI as messages in a mailbox. It analyses each request and charges the CI, which either fulfills the request or translates it into system calls for the OS.
A message from FTI to the same instance on another SBC is handled by the CI too. It converts the message into a standard format, a letter. After that it transmits the letter to the port handler (PH).
To send a letter the PH copies it from the local memory of the sender into the corresponding port (cf. Sec.3) of the receiver. On receiving a letter the PH puts the local time stamp on it and copies it into local memory. The PH manages the logical protocol on the communication channel too. In the OS's view the port handler is a normal device handler. Although our implementation is based on a single communication bus, this concept allows easy switching between different physical communication paths, independent of the bus type used.

4.3 Fault-tolerance instance

FTI implements the fault-tolerance. It communicates only with the CI. Besides for the fault tolerance the FTI is responsible for:
- interpretation of commands to ATTEMPTO
- management of systemwide resources
- control of data streams from/to the peripherals.

Moreover the FTI manages dispatching of user jobs by the fault-tolerant dispatcher (FTD) and fault-diagnosis (Sec. 5). All this is based on information kept in a system table, the system control block (SCB). This table belongs to the FTI only.
In order to have coherent system tables, the same sequence of incoming letters to update the system table must be guaranteed on all SBCs. This is achieved by the protocol described in section 3.2.
To change the system table the local FTI sends its intention via CI to all FTI's including itself. So the decision by each FTI, whether or not the request is granted, is the same.
The dispatching of user jobs is based on the principle of attraction. Contrary to the centralized hardware realization in PLURIBUS /6/ we employ a decen-

tralized software version of this principle. As mentioned, each FTI has its own system control block (SCB). A SCB contains a doubly linked queue of job control blocks (JCB). Each JCB characterizes a job by its name, its security index (SI), the list of the colleagues for this job and their starttimes. SI is the number of colleagues involved; it depends on the tolerance degree t of a job. The JCB-queue is never empty since its last element is the JCB of a resident self-test routine as described in Sec. 5.
The local FTD is invoked under three different conditions:

a) The user has given a command to start a new job. The running job, if any, is interrupted and the FTD examines the command, creates a new entry in the JCB-queue and initiates this entry with jobname and SI. After this, the previously running job, if any, is continued. Otherwise the FTD proceeds as in case b).
b) A job has terminated correctly. First the corresponding entries in the JCB-queue are removed (except for the selftest job). The FTI scans the JCB-queue for JCB's with not yet completely filled colleague-lists. If the FTD has found such a JCB, it tries to attract this job. It sends its intention to start this job to all processors including itself in order to make them update their own JCB. If between sending and receiving its own message no other processor has completed this JCB's colleague list, FTD starts this job. Otherwise it scans the JCB-queue again. By the synchronizing mechanisms described in Sec.3.2 it is guaranteed that messages to update the JCB-queue are sequenced properly.
c) Via interprocessor communication a processor is made to update its JCB-queue. Therefore, the user job is interrupted and the FTD records the time stamp of the received message as starttime in the corresponding place of the starttime-list. The starttime is necessary to set up local time-outs in order to prevent a processor from waiting on messages of a crashed colleague.

The FTI assures correct output in the following way: Before each WRITE-operation all the processors involved in the job (colleagues) have to compare their arguments. Data are compressed by computing signatures (Sec.5). The FTI manages the signature array block (SAB) queue. Each SAB contains an array for all signatures built from output data of a certain job . In this way data structures are established to enable diagnosis.
The actual WRITE-operation is executed by the so-called i/o-master, i.e. that processor which is the first one to start output execution and which received validation (cf. Sec.6.3). The other non-faulty processors watch the correct execution of the output routine.
The procedures of the FTI to compare signatures (recorded in the SAB-queue) and diagnosis are described in the following.

5. Tests and Diagnosis

The unit of fault-localization is a SBC or a bus.
In general, errors are assumed by ATOS to be caused by transient faults (In /8, p.18/ the percentage of transient failures is estimated as more than 90%.). Because of this assumption and because jobs are carried out asynchroneously, it is justified to assume that faulty processors do not compute pairwise the same result. Only if appropriate checks do not confirm the assumption of transient faults, ATOS does consider errors as caused by permanent faults.
Testing of our system is therefore based mainly on the so-called job-result comparison approach /1,3, 7/. In this context, an undesired event occurs if and only if the correct result cannot be identified. Hence provided that not more than t colleagues fail during the execution of a job with tolerance degree t, no undesired event can occur and, accordingly, no recovery action is necessary.
Due to our modular design, a software module implementing conventional testing strategies is readily integrated, should it turn out during validation that one of our assumptions is not justified.
In order to economize the amount of comparison we employ data compression and comparison assignments /1/ which best balance the trade-offs in terms of efficiency, accuracy of fault-localization, and the amount of redundancy used.
During execution of jobs many permanent faults (e.g. certain stuck-at-faults) affect results as transient faults do. Comparison tests detect them as well as transient faults, but do not distinguish between permanent or transient. In order to determine whether or not a detected fault was transient, additional self tests are used. These tests are executed during idle periods (free tests), too. Detection of transient faults by job-result comparison however is very important and cannot be achieved with self-tests only. This approach has further advantages:

- System components are tested by external instances, whereas self-tests require a failure immune hardcore within the components.
- The comparison test approach is conceptually simple and independent of hardware structures; it is portable and flexible.
- While utilizing component redundancy, comparison needs no specialized hardware, only little time overhead, no context switching and it is independent of the failure modes /11/.

Hence, system testing comprises system start-up tests (initial check-out), tests during job execution (job-result comparison and self-tests of faulty processors), free tests and tests for system reconfiguration. These tests implicitely check the communication system on protocol level. Identification of faulty processors is decentralized (cf. /2/) since each processor executes the diagnostic algorithm of Sec.6.3. Thus each processor obtains its own view of the states of some of his colleagues. The view of faultless colleagues is correct. A safety mechanism is provided to prevent a faulty processor to disturb the system (Sec.6.3). If possible, faulty processors should execute self-tests. After that, they can accept new jobs or messages from other processors only if their fault is diagnosed as transient.
Processors record failures of colleagues in fault frequency lists for reconfiguration. If a given threshold is exceeded communication with the concerned processor is dropped.

6. Means for Diagnosis

To determine the states (fault-free or faulty) of the involved SBCs (colleagues), the computation results are first compressed to a feasible normed length (data compression). Then pairs of processors comparing their results are chosen among all possible pairs (data selection). Finally a distributed diagnosing algorithm evaluates the comparison results and completes the local diagnosis.

6.1 Data compression

Using a software version of a linear feedback shift register we compress the results to signatures of a normed length of r bits (in our system r=16= data bus width). Then two data streams differing in exactly one bit lead to different signatures. Hence, all single-bit errors in a data stream are identifiable from the signatures. Furthermore, assume that all possible errors in a data stream are equally probable. Then the probability for identical signatures of the correct and the erroneous data stream is approximately 0.000015 /9, Theorem 3/.

6.2 Data selection

Diagnosis comprises the determination of signatures which are to be compared. Instead of all possible pairs, as few pairs as necessary are selected.
Let signatures correspond to nodes and let comparisons of signatures correspond to undirected edges between two nodes. Then data selection defines an undirected comparison graph G=(N,E) where N is the set of signatures, E⊆NxN the set of comparisons.
Recall that two incorrect results may never agree.
A comparison graph G is called t-diagnosable if all incorrect signatures can be identified, provided their number does not exceed t. G is even t-optimal if in addition its edge number is minimal relative to all t-diagnosable graphs with same node number.
In ATTEMPTO we use t-optimal comparison graphs (see /1/) with $t=\lfloor N \rfloor-3$ (for $t\geq 2$ because i/o-monitoring should be done by at least 3 fault-free processors), where t is the tolerance degree of a job (e.g. 4 in Fig.4). Contrary to the decentralized majority voting, this approach saves O(n) comparisons.

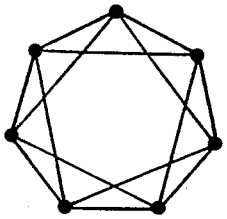


Fig.4 (t=4)

6.3 Diagnosing algorithm

A processor's view of the comparison graph is only local. From t, it computes its neighbourhood in the corresponding graph. (For example, the neighbourhood of processor i is (i+1,..,i+t/2,i-1,..,i-t/2) modulo t+3 for even values of t). All graphs used in ATTEMPTO are even strongly t-optimal for t≥2, i.e. at least two fault-free processors are neighbours (and therefore immediately identifiable as fault-free) and each faulty processor is a neighbour of one of these fault-free processors.
Now each colleague compares its own signature with the signatures of its neighbours. If at least one comparison passes, the processor is sure to be fault-free and correctly diagnoses all of its neighbours. A faulty processor, however, does not find an identical signature among its neighbours. Nevertheless it may diagnose itself as fault-free. In very unprobable special cases the neighbourhood of one fault-free processor concists of all faulty processors. In this case the two remaining fault-free processors have compared themselves (strong diagnosability), perform one more communication step and assure to the isolated processor that it is fault-free. Now, all fault-free processors know that they are not faulty. Faulty processors, however, may have this view of themselves too. In order to prevent faulty processors from performing the output, a second step of mutual inquiry has to follow.

Each processor needs a key (e.g. the start address of the output routine) to begin with the output. This key must be sent to it by another colleague. Therefore, processor i asks a colleague, j, which is fault-free in i's opinion, for the key. Colleague j sends a message back to i, in which the desired key could be found at a location dependent on j's signature. Processor i is able to find the key only if its signature agrees with j's signature. Faulty processors, however, will never find a colleague with the same signature and so will never find the key. Hence, the user is sure to receive output from a fault-free processor.

Conclusion

We have outlined the design concepts of a fault-tolerant working station which is currently being implemented with the intent to meet the increasing demand for fault-tolerant general purpose computing. An important issue for the future will be the validation of our system. In our present implementation certain bus errors can only be detected. Hence, the buses are weak points with respect to fault-tolerance and our present system cannot be classified as ultra-reliable. It should, therefore, be considered as a fault-tolerant computer for office and laboratory tasks which require high dependability and still allow prescheduled maintenance.

This work was supported in part by the German Science Foundation (DFG) under Contract Da 141/2-2.

References

/1/ Amman E., Dal Cin M., Efficient algorithms for comparison-based self-diagnosis, in /4/ pp 1-18
/2/ Ciompi P., Grandoni F., Simoncini L., Distributed diagnosis in multiprocessor systems, Proc. FTCS-11, 1981 pp 25-29
/3/ Chwa K.Y., Hakimi S.L., Schemes for fault-tolerant computing: a comparison of modulary redundant and t-diagnosable systems, Information and Control 49 1981 pp 212-238
/4/ Dal Cin M., Dilger E. (Eds.), Self-diagnosis and fault-tolerance, ATTEMPTO-Verlag, Tübingen 1981
/5/ Färber G., Task-specific implementation of fault-tolerance in process automation, in /4/ pp 84-102
/6/ Katsuki D. et al., PLURIBUS - an operational fault-tolerant multiprocessor, Proc. IEEE, Vol. 66/10, 1978
/7/ Maeng J., Malek M., A comparison connection assignment for self-diagnosis of multiprocessor systems, Proc. FTCS-11, 1981 pp 173-175
/8/ Siewiorek, Swarz , The theory and practice of reliable system design, Digital Press, Bedford, MA, 1982
/9/ Smith J.E., Measurements of the effectiveness of fault signature analysis, IEEE Trans. on Comp. C-29 1980 pp 510-514
/10/ Weinstock C., SIFT: System design and implementation, Proc. FTCS-10, 1980 pp 75-77
/11/ Wensley, J.H. et al., SIFT: design and analysis of fault-tolerant computer for aircraft control, Proceedings of the IEEE, Vol. 66 Oct. 1978 pp 1240-1255
/12/ Wirth N., Modula-2, Springer, Heidelberg 1982