

Hardware-Supported Fault Tolerance for Multiprocessors

Mario Dal Cin, Wolfgang Hohl, Volkmar Sieh

Institut für Mathematische Maschinen und Datenverarbeitung III
Universität Erlangen-Nürnberg, Martensstr. 3, D-91058 Erlangen
hohl@informatik.uni-erlangen.de

Abstract

To provide a computing system to be dependable fault tolerance mechanisms have to be included. Especially massive parallelism represents a new challenge for fault tolerance. In this paper we discuss basic hardware fault tolerance measures for massively parallel multiprocessors and solutions realized for and integrated into different multiprocessor architectures. Further we present our validation technique for dependability based on simulation-based fault injection.

1. Introduction

Dependability and, in particular, reliability are important aspects of the quality of service of complex parallel computing systems used for a wide range of applications in science, industry, and engineering. Users of a parallel computing system expect to have the machine always available, and that they can confide in its results. This reliance can be assured by fault tolerance. Therefore, as failure probability increases with rising number of components and growing system complexity, fault tolerance is an essential attribute of dependable large parallel systems; these systems have to incorporate mechanisms to detect (Section 2) and localize errors (Section 3) as well as mechanisms to reconfigure the system and to recover from erroneous states (Section 4). To evaluate these mechanisms fault injection (Section 5) can be used, since in reality hardware faults do not occur frequently enough to investigate different fault scenarios.

We have developed fault tolerance mechanisms in hardware as well as in software, and have shown their usefulness by implementing them for the multiprocessors MEMSY [15] within the SFB-182 project, and the Parsytec GigaCube [33] within the Esprit project FTMPS. Both machines are used for large-scale scientific and technical computations.

The aim of this paper is to single out those fault tolerance mechanisms which we believe are the most adequate ones for massively parallel systems and which have been investigated by us within the MEMSY and the FTMPS project. Since the presented measures are useful for other parallel architectures as well we will discuss the rationale for our choices and hint at those hardware solutions which we have implemented. However, we refer to [2][15][16][17][18][21][22][31] for a detailed description of this mechanisms and to [39][40][42] for results of our effort to evaluate them.

2. Hardware-based Error Detection

The effective integration of fault tolerance mechanisms into large multiprocessors such as MEMSY or the GigaCube requires a carefully chosen compromise between costs, performance degradation and dependability. For example, high fault coverage and low error latency are required for saving error-free checkpoint data and to avoid error dissemination. This requirement can be fulfilled by hardware-based error detection methods. However, in a large multiprocessor the replication of local error detecting hardware results in huge costs. Moreover, the increased hardware complexity could even increase the probability of faults. On the other hand, software-based error detection has strict limitations as faults affecting both the application and the built-in checking procedures may result in low fault coverage, and a large run-time overhead would drastically reduce the performance.

Therefore, the effective integration of fault tolerance into massively parallel systems requires a proper and cost effective combination of different error detection mechanisms. They can be grouped into the following categories according to the level on which they are applied:

- *Monoprocessor self-checking* techniques in each processing node
- *Central checking* mechanisms for entire clusters of processing nodes
- *Distributed system-wide checking* by mutual tests between nodes.

The fault model applied should cover both permanent and transient faults. Transient faults occur, according to practical experiences, by at least one order of magnitude more often than permanent ones. Fortunately, the most crucial methods and phases in handling both types of faults coincide, such that a general methodology can be developed.

In the *monoprocessor approach* the main emphasis focuses on the reliability of the computing core, as the most intensively utilized resource [22]. However, the basic checks integrated onto the chips alone proved insufficient for the construction of a reliable multiprocessor [35].

A method for checking the computing core more intensively is the *master-checker* setup. The active CPU, the master, drives the system. The checker CPU is synchronized at clock level with the master. It processes the very same program and input data stream as the master. Whenever the master drives an output signal, the checker compares its own value with the data written by the master. A mismatch triggers an error signal. The master-checker mode is supported in many modern microprocessors (like in the MEMSY processor Motorola 88k or Intel Pentium families) by a comparator integrated into the pin driver circuitry, thus reducing the external logic to a few chips for interfacing the error signals.

The master-checker setup assures full fault coverage of uncorrelated errors which affect only a single CPU, but does not detect common mode errors, like the processing of wrong input data. Thus, this method requires additional protection mechanisms for the non-duplicated parts of the system [17]. Moreover, duplication of the processing core is cost intensive and should be avoided for large multiprocessors.

Watchdog processors offer a solution [34]. It has been shown that the majority of processor malfunction results in an incorrect program execution sequence [37]. Thus, the checking of the *program control flow* is of primary importance. A watchdog processor is a coprocessor concurrently monitoring the program control flow either by observing the instruction fetch on the CPU bus or by checking symbolic labels explicitly sent from the main program. The traditional watchdog processor is designed as add-on to a conventional CPU. Thus, the overhead is of the

same order of magnitude as the CPU itself. Another insufficiency of traditional watchdogs is their inability to monitor the cooperation between tasks in a multitasking system and the interactions of the different processors in a multiprocessor.

However, any cost effective extension of a simple monoprocessor solution, like a watchdog processor, to a large multiprocessor should be based on sharing of the additional hardware between clusters of processing nodes. So the relative overhead per node can be kept low. This *sharing* principle can be adopted for *watchdog processors*, if its processing speed is sufficiently high to serve multiple nodes. In [31] program control flow checking for MEMSY was based on this sharing principle and, furthermore, the task was reduced by a novel encoding method. Moreover, this setup is capable of checking the cooperation of different processes even if they are distributed among different processing nodes.

Another means to detect faulty nodes is given by mutual tests between nodes. The most widely used technique is based on the system-wide exchange of <I'm alive> messages (or *heartbeat*). This technique had first been implemented into the fault tolerant multiprocessors of Tandem [6].

3. System-level Diagnosis

The step after error detection is fault diagnosis. Its resolution should allow, in case of permanent faults, the reconfiguration of the system in order to isolate faulty nodes, thus limiting the effect of the fault on performance. For the user, the subsequent recovery can be masked by task redistribution or system reconfiguration. It is only important to know how many fault-free nodes of the system can be used and that none of the possibly faulty nodes remains unregistered.

Due to the large size of multiprocessors and the fact that a node is usually an error containment region diagnosis should be at the level of a processing node. This is called *system-level diagnosis*. In recent years several *distributed*, system-level diagnosis algorithms for multiprocessors were published [9][14][27][30][32] which require, at a first glance, no additional hardware. The aim of the distributed diagnosis is that all fault-free nodes generate identical *correct* diagnostic images of the fault states of all nodes. These algorithms analyze diagnostic information to localize the faulty nodes. After the diagnosis is performed, the fault-free nodes can logically or physically isolate faulty units by stopping to communicate with them.

Developing a scalable and efficient system-level diagnosis algorithm for massively parallel computers, as MEMSY and the Gigacube, implies solving numerous problems. The main diagnosis problem is that faulty nodes can generate incorrect test results and can corrupt and forward diagnostic messages. Furthermore, diagnostic messages can be lost on account of faulty nodes. The diagnosis should also correctly handle situations where faulty nodes block the interprocessor communication and separate several subregions of fault-free nodes. Another problem is the large number of diagnostic messages generated and distributed by each node. Moreover, already tested nodes or links can become faulty during the diagnosis. A diagnosis algorithm has to cope with these situations. As a consequence, diagnosis algorithms have to be fault-tolerant themselves. To make diagnosis reliable and efficient it has been proposed to enhance the multiprocessor by a diagnostic subsystem consisting of diagnosis processors and their interconnection network [41]. This, however is cost intensive. A more cost effective solution is to concentrate the diagnostic functions into a single additional *diagnosis processor per cluster* of nodes with a separate communication network, like in the Parsytec GigaCube machine or the CM-5 [41].

Our approach was to develop a split system-level diagnosis algorithm for multiprocessors. We implemented it on the Parsytec Gigacube in the framework of an Esprit project [18] using the host as supporting hardware. Hence, this algorithm is composed of two parts (see Fig. 1): one part is executed on the multiprocessor (distributed diagnosis) while the other part is executed on the host (centralized diagnosis). The algorithm running on the multiprocessor integrates hardware error detection mechanisms with incomplete coverage. As a result it became event-driven and *conservative*. A diagnosis is conservative if all nodes which detect a neighbor as faulty as well as their tested neighbors which are indicated as faulty by test outcomes are classified as faulty. The diagnosis algorithm minimizes the number of diagnostic messages and tests to be performed. Furthermore, it is based on a realistic diagnosis model, since the number of faults tolerated does not depend on the system interconnection topology [2]. The generated diagnostic image is based on results of hardware-based error detection (Section 2) and on results of an <I'm alive> mechanism [1]. In addition, off-line tests can be initiated by the diagnosis itself.

The centralized diagnosis algorithm running on the host provides an interface to a statistics database, which stores system-wide diagnosis images, and an interface to a recovery mechanism, which reconfigures the system and then rolls back the application. In addition, the developed host diagnosis software classifies correlated errors. These errors cannot be classified locally by the distributed diagnosis algorithm running on the multiprocessor, since knowledge of the global error history stored in the statistics database of the host is required to classify most correlated errors.

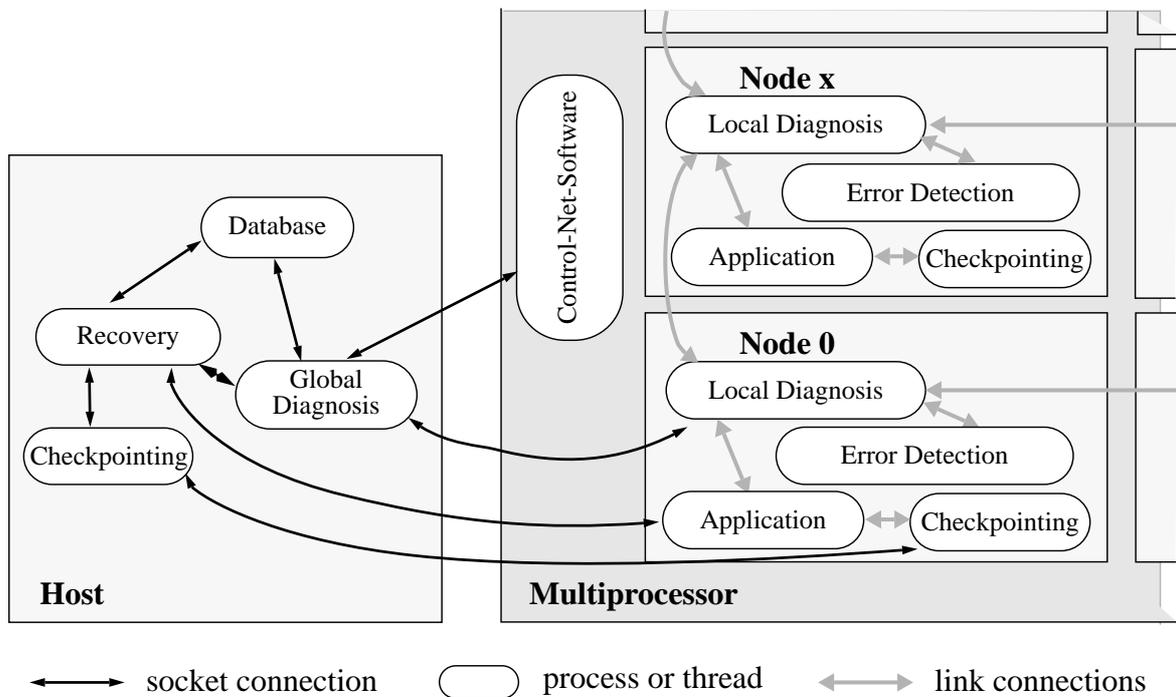


Fig. 1: Fault tolerance modules cooperating with diagnosis

4. Stable Storage and Checkpointing

Checkpointing in conjunction with backward error recovery (roll-back) is a fault tolerance strategy which attempts to restore a previous correct system state after a failure has been detected. This strategy is an attractive solution for massively parallel systems since it does not require additional hardware resources (with the exception of a stable checkpoint storage) as opposed to static fault masking redundancy. Saved checkpoints contain sufficient data to resume the execution of a temporarily failed processing node. If a permanent node failure occurs, it is possible to restart execution on a different node.

For MEMSY, a coordinated global checkpointing scheme based on the notion of *distributed snapshots* has been implemented [11]. The state of a MEMSY application is defined by the state of every participating process and the states of shared memory segments. With distributed snapshots the application processes take checkpoints at about the same time. Shared memory segments are owned by one of the application processes, and are part of the local state of this process. The local checkpoints constitute a global recovery line. In order to ensure consistency of the checkpoint data, communication by messages, shared memory or semaphores is prohibited while a checkpoint is saved. Message logging was avoided since it is quite expensive in shared memory communication as in MEMSY. The checkpoint data is transferred to a stable storage.

As part of the fault-tolerant design effort for MEMSY the checkpointing scheme as well as a stable memory module has been designed [21]. The idea of a stable storage was originally proposed in [28] with a disk-based solution. Paper [13] describes a memory-mapped stable storage and [5] a RAM-based stable storage. Our stable storage is also RAM-based and accessible by a group of processing nodes (sharing principle). Like the stable memory of the Sequoia [8] multiprocessor it uses dual memory banks, A and B. Memory A is updated by normal accesses of a processing node and memory B receives a copy of the data. Our solution is unusual since it contains, in addition, a fault masking control (TMR: triple modular redundancy in Fig. 2).

A prototype of the stable storage has been constructed for MEMSY. It is partitioned into a buffer domain, a central control unit and two memory blocks (object storages) storing stable

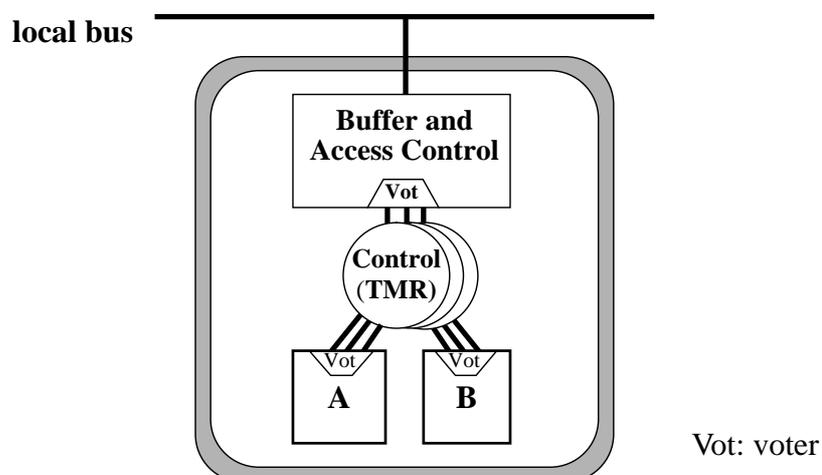


Fig. 2: Stable storage of MEMSY

data objects, e.g. checkpoints. Copies of an object are stored twice in the different memory blocks. With this configuration both an atomic update of a data block can be guaranteed, and the loss of data in case of a single fault can be prevented.

Faults are detected by checksums supplementing each stable object. The transfer (read / write) of stable objects between buffer and object storages and the periodical checking of checksums is executed by the central control unit. The control itself is protected against single internal faults by TMR. Furthermore, the control checks whether a given processing node is allowed access to a specific stable object in the stable storage. In fact, to prevent faulty processors from damaging objects they are not able to access the stored objects directly. They only have access to the buffer domain.

The TMR-control consists of three Transputers 805 with own memories for program and data. They are started synchronously and execute the same program. However, while accessing their local memory they run independently. Only when accessing the buffer domain their addresses and their data are compared for voting.

Before accessing the stable storage the first time, a process obtains by the control a password and an address of a free communication page. All further accesses are executed by reading and writing within this communication page. At a read request the central control unit starts transferring the first copy. If it detects an error, a transfer of the back-up copy follows, and subsequently the central control performs a cleanup. Handling of a write request is more complicated. There is the danger that two independent errors arise at the same time, e.g., a bit flip in a stored object and a faulty data transmission. Hence, at first one copy, copy A say, is checked. If it is faultfree, this copy is taken as backup. During the write phase the other copy, copy B, will be used. On success, the backup copy A is updated. If an error is detected during the first phase, the update can be undone by overwriting copy B with the backup copy A. If the first examination results in a damaged copy A, copy B will be checked. If it is faultfree, it is used as backup copy. If copy B is damaged too (i.e., there are at least two errors within this object), an "extended cleanup" will be performed. This procedure restores the correct object by extracting faultless parts from both copies. This is possible because each word is transferred and stored together with four parity bits (one bit per byte). If the stable storage detects a transmission error (bit flip) by its parity checkers it signals a bus error to the processing node and no other action is executed.

In evaluating our approach the most interesting question was, how much time a user process needs for transferring data into the stable storage because this time determines the overhead for storing checkpoints [21].

5. Fault Injection for Evaluation of Fault Tolerance Mechanisms

Because of the complexity of multiprocessors as MEMSY or the GigaCube it is very difficult to determine the failure rate of such systems even if the fault rate of all the components is roughly known. Therefore, to estimate the dependability of these systems it is necessary to test their behavior under faults. Due to the fact that faults will happen infrequently their appearance has to be enforced by fault injection. Otherwise no statistical analysis can be performed.

In recent years several approaches to inject faults into running systems have been published [3][12][23][26]. Injecting faults at the *physical level* has been done either by stressing the hardware with environmental parameters (heavy-ion radiation, power disturbances, electromagnetic

interference) [19] or by modification of the pin-level values of internal components [29]. Both approaches of the injection at the physical level tend to have a high overhead in hardware and are only feasible after the system has already been produced at least as prototype. Therefore it is not possible to evaluate the system's reliability at early design phases. *Software implemented fault injection* (SWIFI) also needs the physical hardware to inject faults. Several research groups have developed powerful tools to inject faults by software [7][10][25][36][38]. The task's memory image and register set can be corrupted during run-time. Another drawback of SWIFI is the fact, that they cover only a subset of an unknown size of all possible system faults, which makes it difficult to evaluate the reliability of the system. The major advantage of *simulation based fault injection* [24][38] is the observability of all components, which have been modeled. Therefore, this approach simplifies the evaluation of dependability because it covers almost all levels of abstraction. The other benefit of simulation based fault injection is the ability to obtain values for reliability already at the design phase of the system as the physical hardware is not needed for this method. The only disadvantage of this kind of fault injection is the massive overhead in simulation time which has to be expected.

Because of the advantage of accuracy and observability we decided to use the simulation based fault injection approach. Fig. 3 depicts the architecture of our evaluation tool VERIFY (VHDL-based Evaluation of Reliability by Injecting Faults efficientlY) [39][40]. The most important questions regarding fault tolerance are whether a system is able to recover from a specific kind of fault, and how long it will take until the system is in a correct state again. Our tool delivers the answer to this question automatically. To improve simulation efficiency we applied enhancements presented in [20]. All system components have to be modeled in the VHDL hardware description language [4]. By using a standard synthesis tool this description on behavior level can be converted to a gate-level description. Our tool uses this description and a slightly extended cell library to evaluate, e.g., fault latency and fault coverage automatically.

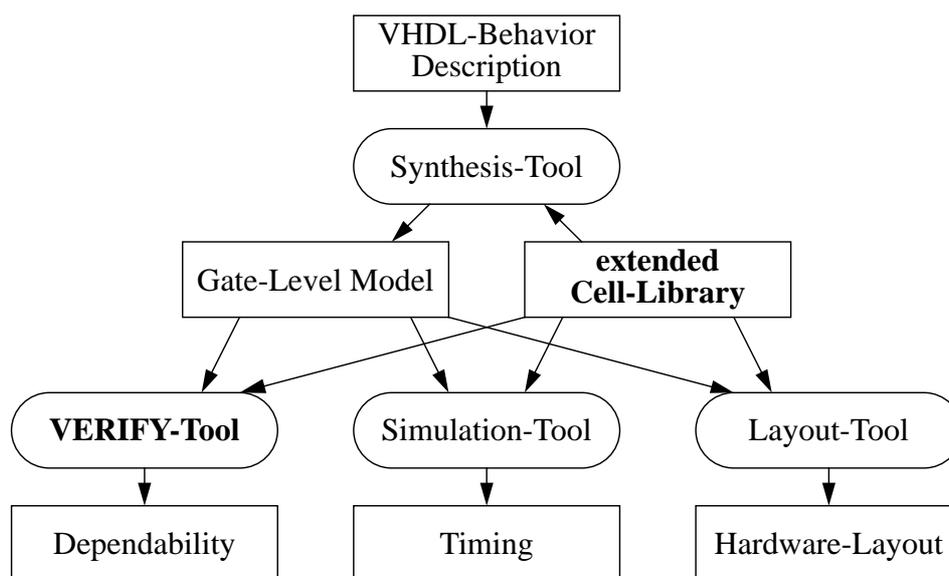


Fig. 3: Dependability evaluation using VERIFY

6. Conclusion

Any massively parallel system is rich of resources and will experience component failures from time to time. The designers of massively parallel systems cannot just demand that no parts of the system should fail, they have to be aware that this or that part of the system may eventually fail and have to take the steps to add redundancy and reconfigurability. It appears that the designers of massively parallel computers have recognized this fact. Suitable measures to tolerate hardware faults and to enlarge the dependability of parallel systems, their realization, and their integration into different architectures have been discussed.

Acknowledgments

The authors thank the Deutsche Forschungsgemeinschaft (DFG) and the European Union (EU) for supporting this work as part of the SFB 182, and the Esprit Project 6731 - FTMPS (A Practical Approach to Fault-Tolerant Massively Parallel Systems), resp.

References

- [1] J. Altmann, F. Balbach, and A. Hein, An Approach for Hierarchical System Level Diagnosis of Massively Parallel Computers Combined with a Simulation-Based Method for Dependability Analysis, in: Proc. *EDCC-1*, Springer LNCS, Vol. 852 (1994) 371-385.
- [2] J. Altmann, T. Bartha, and A. Pataricza, On Integrating Error Detection into a Fault Diagnosis Algorithm for Massively Parallel Computers, in: Proc. *IPDS'95, IEEE International Computer Performance and Dependability Symposium* (1995) 154-164.
- [3] J. Arlat, Fault Injection for the Experimental Validation of Fault-Tolerant Systems, in: Proc. *Workshop Fault-Tolerant Systems*, Kyoto, Japan, IEICE, Tokyo (1992) 33-40.
- [4] P. Ashenden, *The VHDL-cookbook*, Technical Report, Univ. of Adelaide, South Australia, 1990.
- [5] M. Banâtre, G. Muller, B. Rochat and P. Sanchez, Design Decisions for the FTM: A General Purpose Fault Tolerant Machine, in: Proc. *FTCS-21* (1991) 71-78.
- [6] J.F. Bartlett, A 'Non-Stop' Operating System, in: Proc. *Hawaii Intl. Conf. of System Sciences*, 1978.
- [7] J. Barton, E. Czeck, Z. Segall and D. Siewiorek, Fault Injection Experiments using FI-AT, *IEEE ToC*, Vol. 39, No. 4 (1990) 575-582.
- [8] P.A. Bernstein, Sequoia: A Fault-tolerant Tightly Coupled Multiprocessor for Transaction Processing, *Computer*, Vol. 21, No. 2 (1988) 37-45.
- [9] R. Bianchini and R. Buskens, An Adaptive Distributed System-Level Diagnosis Algorithm and its Implementation, in: Proc. *FTCS-21* (1991) 222-229.
- [10] J. Carreira, H. Madeira and J. G. Silva, Xception: Software Fault Injection and Monitoring in Processor Functional Units, in: Preprints *DCCA-5, Dependable Computing for Critical Applications*, Urbana Champaign, 1995, 135-149.

- [11] C.M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM To CS*, Vol. 3, No. 1 (1985) 63-75.
- [12] J.A. Clark and D.K. Pradhan, Fault Injection - A Method for Validating Computer-Systems Dependability, *Computer*, Vol. 28, No. 6 (1995) 47-56.
- [13] B.A. Coghlan and J.O. Jones, Stable Memory for a Disk Write Cache, *Microprocessing and Microprogramming*, Vol. 41, No. 1 (1995) 53-70.
- [14] M. Dal Cin and F. Florian, Analysis of a Fault-Tolerant Distributed Diagnosis Algorithm, in: Proc. *FTCS-15* (1985) 159-164.
- [15] M. Dal Cin, H. Hessenauer and W. Hohl, The Modular Expandable Multiprocessor System MEMSY, *Computer Systems Science & Engineering*, Vol. 11, No 4 (1996) 211-219.
- [16] M. Dal Cin, W. Hohl, J. Hönig and A. Pataricza, MEMSY - A Modular Expandable Multiprocessor System with Fault Tolerance, in: Proc. *Parallel Systems Fair* of the 8th IEEE Int. Parallel Processing Symposium, Cancun, 1994, 21-28.
- [17] M. Dal Cin, W. Hohl, E. Michel and A. Pataricza, Error Detection Mechanism for Massively Parallel Multiprocessors, in: Proc. *EUROMICRO Workshop on Parallel and Distributed Processing* (1993) 401-408.
- [18] G. Deconinck, J. Vounckx, R. Lauwereins, J. Altmann, F. Balbach, M. Dal Cin, J.G. Silva, H. Madeira, B. Bieker, and E. Maehle, A Scalable Implementation of Fault Tolerance for Massively Parallel Systems, in: Proc. *MPCS'96, 2nd Intl. Conf. on Massively Parallel Computing Systems* (1996) 214-221.
- [19] U. Gunneflo, J. Karlsson, and J. Torin, Evaluation of Error Detection Schemes using Fault Injection by Heavy-Ion Radiation, in: Proc. *FTCS-19* (1989) 340-347.
- [20] J. Güthoff and V. Sieh, Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method, in: Proc. *FTCS-25* (1995) 196-206.
- [21] A. Grygier and M. Dal Cin, A Stable Storage Unit for Multiprocessors, in: Proc. *Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, Ca, 1995, 158-163.
- [22] W. Hohl, E. Michel and A. Pataricza, Hardware Support for Error Detection in Multiprocessor Systems - A Case Study, *Microprocessors and Microsystems*, Vol. 17, No.4 (1993) 201-206.
- [23] R.K. Iyer, Experimental Evaluation, in: Proc. *FTCS-25* (1995) 115-132.
- [24] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson and J. Karlsson, Fault Injection into VHDL Models: The MEFISTO Tool, in: Proc. *FTCS-24* (1994) 66-75.
- [25] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, FERRARI: A Tool for the Validation of System Dependability Properties, in: Proc. *FTCS-22* (1992) 336-344.
- [26] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson and U. Gunneflo, Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms, *IEEE Micro*, Vol. 14, No. 1 (1994) 8-23.

- [27] J.G. Kuhl and S.M. Reddy, Fault-diagnosis in Fully Distributed Systems, in: Proc. *FTCS-11* (1981) 100-105.
- [28] B.W. Lampson, Atomic Transactions, in: M. Paul and H.J. Siegert (eds.), *Distributed Systems - Architecture and Implementation*, Springer LNCS, Vol. 105 (1988) 246-265.
- [29] H. Madeira, M. Rela and J. G. Silva, RIFLE: A General Purpose Pin-Level Fault Injector, in: Proc. EDCC-1, Springer LNCS, Vol. 852 (1994) 199-216.
- [30] E. Maehle and H. Joseph, Selbstdiagnose in fehlertoleranten DIRMU-Multi-Mikroprozessorkonfigurationen, in: Fehlertolerante Rechnersysteme, *Informatik-Fachberichte*, No. 54 (Springer, 1982) 59-73.
- [31] I. Majzik, W. Hohl, A. Pataricza and V. Sieh, Multiprocessor Checking using Watchdog Processors, *Computer Systems Science & Engineering*, Vol. 5 (1996) 301-310.
- [32] F.J. Meyer and G. Masson, An Efficient Fault Diagnosis Algorithm for Symmetric Multiprocessor Architecture, *IEEE ToC*, Vol. EC-27 (1978) 1059-1063.
- [33] Parsytec Computer GmbH, *The Parsytec GC Technical Summary*, Version 1.0, Aachen (Germany), 1991.
- [34] A. Pataricza, I. Majzik, W. Hohl and J. Hönig, Watchdog Processors in Parallel Systems, *Microprocessing and Microprogramming*, Vol. 39 (1993) 69-74.
- [35] M. Rela, H. Madeira and J.G. Silva, Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks, in: Proc. *FTCS-26* (1996) 394-403.
- [36] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson and T. Lin, FIAT — Fault Injection Based Automated Testing Environment, in: Proc. *FTCS-18* (1988) 102-107.
- [37] M. Schmid, R. Trapp, A. Davidoff and G. Masson, Upset Exposure by Means of Abstraction Verification, in: Proc. *FTCS-12* (1982) 237-244.
- [38] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Hönig and B. Benyo, Fault Injection Based Validation of Fault-Tolerant Multiprocessors, in: Proc. $\mu P'94$, *8th Symposium on Microcomputer and Microprocessor Applications* (TU Budapest, 1994) 85-94.
- [39] V. Sieh, O. Tschäche and F. Balbach, Comparing Different Fault Models using VERIFY, in: Proc. *DCCA-6, Dependable Computing for Critical Applications* (1997) 59-76.
- [40] V. Sieh, O. Tschäche and F. Balbach, VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Description, in: Proc. *FTCS-27* (1997).
- [41] Thinking Machines, *CM-5 Technical Manual*, 1992.
- [42] J. Vounckx, G. Deconinck, R. Lauwereins, G. Viehöver, R. Wagner, H. Madeira, J.G. Silva, F. Balbach, J. Altmann, B. Bieker, and H. Willeke, The FTMPs-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems, in: Proc. *HPCN Conference*, Springer LNCS, Vol. 797 (1994) 401-406.