Application Dependent Performability Evaluation of Fault-Tolerant Multiprocessors

S. Dalibor, A. Hein, W. Hohl

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

Abstract

A case study of performance and dependability evaluation of fault-tolerant multiprocessors is presented. Two specific architectures are analyzed taking into account system functionality, actual workloads, failures of system components as well as the inter-component dependencies. Since the evaluation of such complex systems has to be performed already during the design phase, simulation models are developed and used to provide insight into the system behavior and to uncover weak points and bottlenecks. Object-oriented software design and process-oriented simulation techniques are used for model construction allowing sophisticated performance and dependability analysis of massively parallel systems.[1]

Keywords: performability analysis, reconfiguration policies, workload modeling, simulated fault injection

1. Introduction

Multiprocessors become more and more important to solve computationally extensive and time-consuming problems in science, industry, and engineering. Since failure probability increases with rising number of components, fault tolerance is an essential characteristic of massively parallel systems; they have to provide redundancy and mechanisms to detect and localize errors as well as to reconfigure the system and to recover from error states. These fault tolerance mechanisms are typical features of the so-called *gracefully degrading systems* that continue working with deteriorated performance if components fail.

Computer systems have to be evaluated as early as possible in order to compare alternative design approaches and to facilitate the change of a current design, i.e., methods are needed to analyze the target systems already during the early design phase. Since the real system is not available for benchmarking or monitoring in this stage, the system designer has to rely on theoretical methods such as analytical and simulation models.

To reflect the important features of multiprocessors we have developed simulation-based models of the target systems including their essential functionalities (e.g. the scheduling and routing mechanisms), actual workload, and fault tolerance mechanisms. Number-crunching algorithms solving scientific problems represent typical workload for the target systems. Additionally, fault tolerance mechanisms developed for the real multiprocessors are implemented and tested via the simulation models.

We want to emphasize the fact that performance and dependability are not analyzed separately, but a performability analysis - a combined performance and dependability analysis - of the overall system is conducted (Fig. 1). In Chapter 2. the modeling approach is discussed. The target systems and their simulation models are presented in Chapter 3. The systems are analyzed in Chapter 4. and a summary including an outlook is given in Chapter 5.

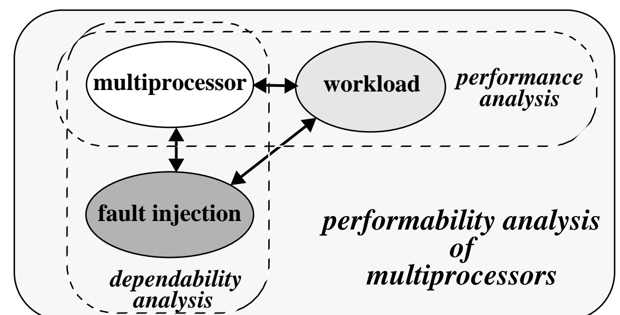


Fig. 1 Performability Analysis

2. The Modeling Approach

Simulation models of the target architecture are powerful means delivering accurate results and avoiding some constraints of *analytical modeling* such as state-space explosion and unrealistic simplifying assumptions, for instance the ‘memoryless’ property in Markovian models. In comparison to *Petri net-* and *Markov*-based analytical models, simulation-based models offer greater accuracy

1. This paper is submitted to Euromicro’s *4th Workshop on Parallel and Distributed Processing* PDP ’96, Braga (Portugal), January 1996

and flexibility. Surveys of different modeling techniques and various tools can be found in [6], [11].

In order to perform a sophisticated analysis of fault tolerant multiprocessors the system-level simulator *SimPar* has been developed for performance and dependability evaluation [5]. To implement *SimPar* the *Depend* tool [3] is taken as the underlying simulation engine which provides basic components such as simulation classes for fault-tolerant servers and link connections. It has successfully been used for the analysis of a TMR-based system and for the simulation of software behavior under hardware faults [4]. *SimPar* provides new enhancements to facilitate the model development and the performance and dependability analysis of massively parallel fault tolerant systems.

The main characteristics of the modeling approach are *process-oriented simulation* and *object-oriented software design*. Another essential feature is the *fault injection* into components of the simulation model in order to interrupt and disturb their regular and predefined behavior.

The behavior of the system is modeled by light-weight processes. The *process-oriented approach* provides an intuitive model development in depicting the system from the point of view of a participating entity. Therefore, it is well-suited for the design of large-scale models and for the modeling of complex inter-component dependencies. A comprehensible introduction into the technique of *process-oriented simulation* can be found in [10]. The overall system behavior is described by a collection of asynchronously interacting processes. Processes depict the functionalities of the system components such as processors and switches. Besides, the process-oriented simulation environment allows actual user-defined distributed programs (written in C or C++) to be executed within the simulation model to perform detailed analysis and to test the efficiency of implemented fault tolerance algorithms. These programs represent user-defined workload such as distributed numerical algorithms as well as system software for routing and reconfiguration tasks.

Object-oriented design is the construction of a software system as a structured collection of abstract data type implementations [8]. Data abstraction and inheritance are central features of the object-oriented software design of large-scale models representing massively parallel systems [7]. Reusability and scalability of the models are supported by the possibility of inheritance and derivation. In several phases of inheritance and derivation, very powerful and specialized classes can be provided by keeping an hierachical and clear model design without reimplementing every class from scratch. Additionally, the class hierarchy facilitates the reusability and maintainability of complex simulation models by hierarchical model construction and step-by-step refinement.

In accordance with the *object-oriented software design* [8] classes are implemented in the object-oriented language C++ encapsulating the functionalities of the processors, switches, and links. Objects of the simple classes are combined to the highly complex structures of multiprocessors. When the model of the target multiprocessor is initialized, objects of these classes are instantiated and dependencies between the objects are defined depicting the characteristics of the real system. The physical connections between the processors, switches, and links are simulated via logical dependencies between corresponding objects. The model can easily be modified, for example, by replacing the current class of the processor by another class, simulating the real processor in more detail or representing a completely different type of processor.

Faults can be injected in objects of every basic class, disturbing and interrupting the predefined fault-free behavior in order to examine the influence of faults on the overall system. The model designer can define his or her own fault model by assigning functions to the basic classes which are called as soon as faults are injected; if no such fault function is defined, the implemented default fault model is assumed which is outlined in Section 3.3.

3. The Target Architectures and their Model Representation

In this chapter we describe the architectures of two multiprocessors as well as the respective simulation models.

3.1 Distributed Shared-Memory Architecture

MEMSY (Modular Expandable Multiprocessor System) is a massively parallel multiprocessor architecture developed at the University of Erlangen-Nürnberg for large-scale scientific and technical computations, based on the concept of distributed shared communication memory [1], [2]. Its principal design goal is to enhance the shared memory paradigm with the possibility of unlimited scalability; this is achieved by the arrangement of processing nodes in an easily extensible, regular structure. Only the communication memory and the interconnection network for accessing the shared memory modules consist of specialized hardware. The processing nodes are built from state-of-the-art microprocessors.

The processing nodes form a topology with constant local interconnection complexity: Toroidal grids of tightly coupled nodes are stacked so that each node of an upper level has access to the communication memories of the four nodes directly below. As the number of nodes of each level is four times smaller than the number in the grid below, the global structure resembles a pyramid. Scalability

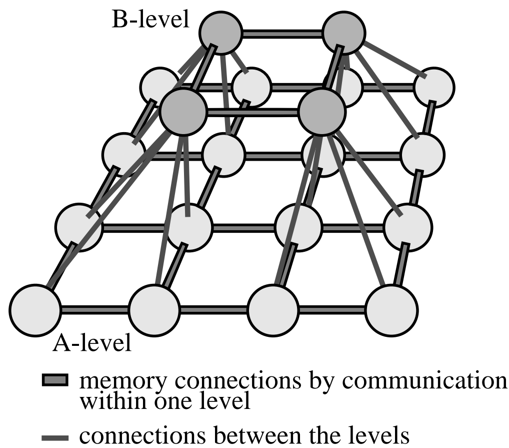


Fig. 2 MEMSY Topology

is possible due to the constant local connectivity: The structure can be extended by adding more levels of grids or by enlarging the number of nodes per grid. An experimental prototype of MEMSY is currently operating; it contains 20 (4+16) nodes in two levels (see Fig. 2; the toroidal connections are not shown). Each processing node consists of a standard Motorola MVME188 board system (with four MC88100 RISC CPUs) plus additional hardware for the memory interconnection.

The fact that each MEMSY-node is a full-fledged parallel computer is also reflected by the operating system MEMSOS: Per node a standard UNIX System V kernel is running with symmetric multiprocessing capabilities (referring to the four CPUs) as well as extensions for access to the communication memory, inter-node-interrupts, and management of applications with processes distributed over various nodes. Because the nodes are running a multiuser, multi-tasking operating system, and are accessible through standard network services like Telnet and Remote Login, it is not unusual that some or all nodes are shared between several distributed applications and other user-oriented processes like editors or compiler jobs. In general, this is not a problem because four CPUs are available per node. However, from the viewpoint of an application it is possible that one or more nodes participating in computing are slowed down due to heavy load and the fact that application processes have no special priority compared to other UNIX-processes running on a node. The impact of such performance degradation of the nodes involved in an application is investigated in Section 4.1.

3.2 Message-Passing Architecture

The target message-passing architecture is a Parsytec GC multiprocessor [9]. This massively parallel machine is designed for scientific and technical applications requiring huge computing power. High performance is achieved by the large number of processing elements and their interprocessor connections providing high bandwidth. In addition, this system takes into account the increasing probability of a failure of system components: Redundant processors may replace faulty ones and the communication paths are redundant to reduce communication delays as well as to tolerate failures of links and switches. The architecture consists of a *data network* forming a three-dimensional grid on which the user application runs. The nodes are connected to every neighbouring node via 8 communication links. Each of the nodes corresponds to a cluster containing 17 INMOS T9000 processors which are redundantly connected via four crossbar-like INMOS C104 routing switches (Fig. 3). Only 16 out of the 17 processors of a cluster are used for application programs; the 17th processor serves as spare component. Besides, an additional network - the so-called *control network* which is not shown in the figure - supervises and controls the components of the data network.

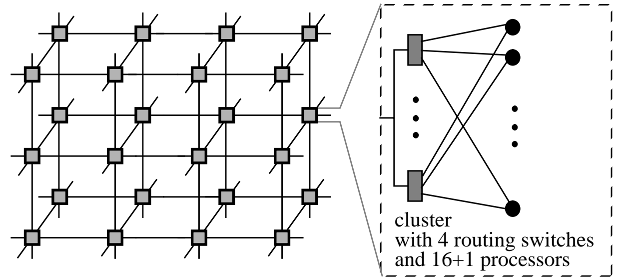


Fig. 3 Data Network of the Parsytec GC

3.3 Simulation Models

The following three model components are the basic constituents of the overall simulation model providing the essential functionalities of the target components as well as default fault models. The model designer can define another fault model for all the basic objects by simply writing functions which are called when a fault is injected:

- The *processor object* P is supposed to perform processes, to schedule them, and it has to receive and send messages. If the model designer does not provide another fault model, P ejects any jobs in progress and stops handling messages as soon as a fault is injected.
- The *routing switch object* R receives and forwards messages in accordance with a predefined routing scheme. A faulty switch no longer communicates with its links, and all messages buffered in the switch are lost.
- The *link object* receives messages at a port and forwards them to another port. When a link object is injected by a fault, it stops forwarding messages from the incoming to the outgoing port or the messages are corrupted.

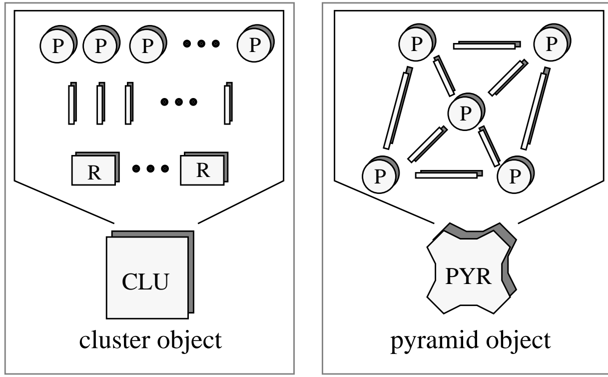


Fig. 4 Basic Objects of the Simulation Models

The relationships and dependencies among the basic components of the simulation environment represent the topology and functionality of the target system. Based on the three basic components more complex objects are created which model parts of the multiprocessor such as the clusters of the Parsytec GC consisting of 17 processor objects and 4 routing switch objects connected by link objects (object CLU in Fig. 4 and Fig. 5).

Similarly, 5 processor objects and several link objects are combined in an object representing the smallest unit of the MEMSY topology (object PYR in Fig. 4 and Fig. 6).

After the user has defined the number of clusters in the three spatial dimensions of the data network, the required set of cluster objects CLU is initialized and the topology of the Parsytec GC is automatically modeled by defining the communication paths between the clusters. The model of MEMSY is initialized in a similar way; the user inputs the number of basic pyramids in the two dimensions, and the according pyramid objects PYR are initialized and put together. The final objects MP_{GC} and MP_M are models of the overall target multiprocessors (Fig. 5 and Fig. 6).

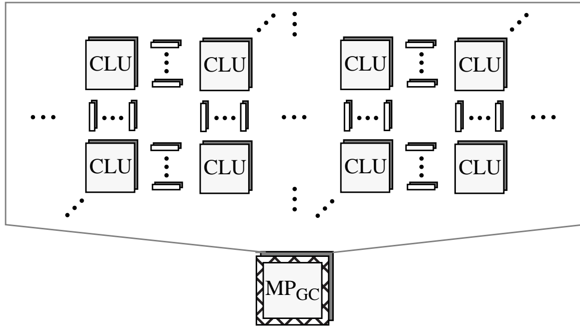


Fig. 5 Object MP_{GC}: Model of the Parsytec GC
(only two of the three dimensions are sketched)

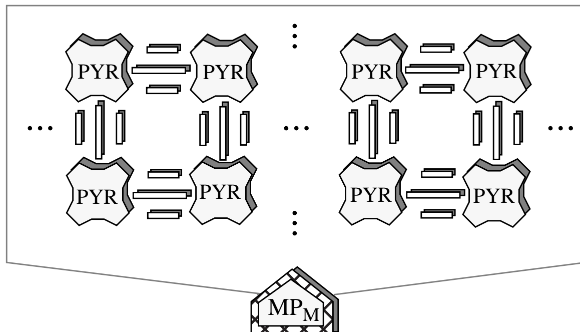


Fig. 6 Object MP_M: Model of MEMSY

The multiprocessor objects MP_{GC} and MP_M are totally scalable by varying the number of clusters and pyramids in the three respectively two dimensions of the networks. These objects are the interfaces to the user-written control program, i.e., the user has to call methods of these objects to load user-defined processes on the processors. These processes can be real distributed application programs as well as operating system routines. Besides, programs to perform fault diagnosis and reconfiguration of the simulated multiprocessor can run concurrently to distributed number crunching algorithms modeling realistic workload. In the Parsytec GC model MP_{GC} messages are sent from the sending processor object through the intermediate link and switch objects of the network to the receiving processor object. As in the target system, successfully received messages are acknowledged.

Additionally, the MP_{GC} and MP_M objects provide methods to define the fault injection, to get information about the current system state, and to output fault reports. The system analyser defines and controls the experiments by simply calling methods of these objects.

4. Results

In this section we present the two specific analyses of our target multiprocessor architectures performed by using the *SimPar* simulation models.

4.1 Impact of Node Performance Degradation on Execution Times on MEMSY

Each MEMSY-node consists of four CPUs - but from the viewpoint of the application programmer they are considered as one computational resource. A MEMSY application takes full advantage of the computing power of a node by creating several UNIX-processes per node; as long as no other jobs are started on the node, up to 4 processes are running completely parallel. Because accesses to the communication memory are satisfied directly by the hardware (i.e. the memory management units and the system bus), the load generated by the operating system is negligible after an initialization phase. If however more than four processes per node have to be scheduled, the op-

erating system distributes the CPUs in a time sharing manner among the processes. With regard to a single application, this can be considered as *performance degradation* of the node: For each additional process started, the computing power available for one specific application is diminished.

In order to investigate the impact of the performance degradation of one or more nodes due to application-external load under controlled conditions, a typical distributed algorithm solving the many-body-problem has been executed on the MEMSY simulation model. This algorithm requires a complete information exchange between all nodes after every computing phase; taking into account the distributed memory structure of MEMSY, this is realized as distributed broadcast exploiting the toroidal connection within the grid-level. Between the computing phases, the processes exchange data packets of varying size with neighboring nodes via their communication memories by performing ring-shifts in horizontal and vertical direction until the information from all the other nodes is available at each node of the grid.

The influence of application-external load was simulated by decreasing the number of CPUs working for the application; the following cases were taken into account:

- *exp_0*: no external load, all CPUs working for the application.
- *exp_1*: 1 CPU on 1 node was taken from the application.
- *exp_2a*: 2 CPUs on 2 different nodes were taken from the application.
- *exp_2b*: 1 node was working for the application with only 2 CPUs.
- *exp_3a*: 3 CPUs in 3 different nodes were withdrawn from the application.
- *exp_3b*: 2 CPUs in 1 node and 1 CPU in another node were unavailable for the application.
- *exp_3c*: in 1 node, only 1 out of the 4 CPUs was working for the application.

The experiments have been conducted with two different message sizes on a system with four basic pyramids; the results are shown in Table 1. In the experiment with larger message size, the total run time is less increased - this can be explained by the fact that the nodes are exchanging messages for the most part of the time and the reduced node performance has no significant impact on the overall run time. The experiments have shown clearly that the overall run time of the application depends to a high degree on the slowest node involved in the computation - it is for example considerably worse to have only 2 CPUs in 1 node (*exp_2b*) participating in the application than 3 CPUs in 3 different nodes (*exp_3a*). Besides, the impact of the application-external load is not at all negligible: Deterioration of the total run time of more than 30% if only 1 node of the A-level contributes to the application with 3 instead of 4 CPUs indicates that measures should be taken to compensate for the additional load or to restrict the number of the additional processes (see Table 1).

	small messages		large messages	
run time	absolute	relative	absolute	relative
exp_0	22.2286	1.0	264.158	1.0
exp_1	30.2822	1.3623	265.158	1.0038
exp_2a	31.1584	1.4017	265.958	1.0068
exp_2b	38.7822	1.7402	267.283	1.0118
exp_3a	31.5528	1.4195	266.587	1.0092
exp_3b	38.7822	1.7447	268.083	1.0149
exp_3c	63.7822	2.8694	276.742	1.0476

Table 1 Results of the Simulation Experiments

Referring to the operating system MEMSOS, a special run-queue with increased priority for the processes involved in distributed applications could be a possibility to avoid exaggerated slowdown of computations. Another method currently under investigation is the utilization of the B-level-nodes as ‘spare-processors’ - if one of the A-level-nodes gets overloaded to an intolerable extent, the belonging B-level-node could either ‘lend’ one or more of its CPUs or take over the task of the loaded node completely. The integration of adaptive load-balancing into the operating system seems to be desirable - future experiments with the MEMSY simulation model MP_M will be useful to investigate possible strategies.

4.2 Evaluation of Reconfiguration Policies in a Message-Passing Architecture

The Parsytec GC system is analyzed by injecting faults into the processors of the *SimPar* model. After error detection and localization of the faulty component, the system has to be reconfigured, i.e., spare processors replace faulty processors and continue their tasks. Each of the clusters contains a 17th processor as spare (cf. Fig. 3). As the user is shielded from the actual network topology and the system is reconfigured automatically, a reconfiguration policy has to be specified and implemented in order to determine the spare processor to replace the faulty processor dependent on the state of the overall system.

Loss of performance in terms of longer execution time can be caused by the following reason: Processes running

on physically neighboring processors exchange messages relatively fast. After a reconfiguration these processes are possibly assigned to not-neighboring processors, i.e., they belong to different clusters of the multiprocessor; communication between processes assigned to different clusters cause time-consuming intercluster data exchanges increasing the overall run time of the application program.

Different reconfiguration policies are considered and compared with regard to application programs with various communication patterns. Identification numbers are assigned to the clusters ranging from *0* to *number_of_ clusters-1*. We assume that a processor fails in cluster *fcl* and a spare processor is selected in cluster *scl*; fcl_x, fcl_y, and fcl_z are the grid coordinates of cluster *fcl*. The distance *d* between two processors corresponds to the distance of the clusters the processors belong to; *d* is defined as:

$$d = |fcl_x - scl_x| + |fcl_y - scl_y| + |fcl_z - scl_z|$$

The following six reconfiguration policies are evaluated in the experiments described in this chapter:

- *rec_a*: The spare processor is chosen according to the *internal order* of the clusters. If the spare processor of cluster 0 is intact and free, it replaces the faulty processor; otherwise, if the spare processor of cluster 1 is still intact and free, it is selected; otherwise the spare processor of cluster 2 is checked, etc.
- *rec_b*: The spare processor of cluster *fcl* takes over the processes of the faulty processor, if it is fault-free and unused. Otherwise, the clusters *fcl+1, fcl+2, fcl+3,etc.* are searched for an available spare processor.
- *rec_c*: The cluster is *randomly* chosen until a cluster *scl* with an unused and intact spare processor is found to replace the faulty one.
- *rec_d*: The spare processor of cluster *fcl* replaces the faulty processor (*scl = fcl*), if it is fault-free and unused. Otherwise, a spare processor is selected in the cluster *scl* with *minimal distance d* to cluster *fcl*.
- *rec_e*: Policy *rec_e* is a variant on *rec_c*: Again, the spare processor of cluster *fcl* replaces the faulty (*scl = fcl*), if it is fault-free and unused. Otherwise, another cluster is *randomly* chosen which still contains a spare processor.
- *rec_f*: This policy is the contrary of *rec_d* and is used for comparison purposes. After the failure of a non-spare processor of cluster *fcl*, *rec_f* determines a cluster *scl* which contains a spare and has *maximum distance d* to cluster *fcl*, i.e., the communication paths become as long as possible.

Simulating a system with 8 clusters, the processes are mapped on the processors and the applications are running on the 128 processing nodes. First, we have measured the mean run time T_0 of the distributed application without processor failures; this times are compared with mean run times T_k after *k* processors have failed, and after fault diagnosis, reconfiguration, and recovery have been carried out. The results presented are given in *ratio of mean run time* $R_k = T_k/T_0$. In the optimal case, at most one processor per cluster fails and is replaced by the spare processor of the same cluster keeping short communication distances. A primitive reconfiguration algorithm not considering this issue of close or long communication distances increases the communication overhead. Additionally, if more than one processor per cluster fails, spare processors of other clusters, which are selected in accordance with the predefined reconfiguration policy, have to take over the processes.

Most of the numerical algorithms implemented on multiprocessors can be characterized by a loop containing a computation phase and a communication phase for data exchange:

```
while (result accuracy is not sufficient) {
  compute();
  /* phase I: compute on private data */

  communicate();
  /* phase II: exchange data */
}
```

Phase II is responsible for the communication overhead and can have a large impact on the overall execution time dependent on the message handling, the topology of the multiprocessor, the size and the frequency of the messages. We started algorithms with typical communication patterns on the simulator MP_{GC} and analyzed the impact of the reconfiguration policies on the execution times.

4.2.1. Linear Array Communication

A numerical algorithm is started on the simulated multiprocessor, whose communication pattern is a linear array, i.e., process *i* exchanges data with process *i+1* and process *i-1*. This pattern is very common in parallel computation of partially differential equations; the data is partitioned among the processes and the border areas of the data have to be exchanged between logically neighboring processes. For this communication pattern it is straightforward to construct an optimal mapping scheme allowing shortest communication time and therefore shortest run time of the distributed algorithm (Fig. 7a). As soon as a processor fails and the failure is detected, the system is reconfigured, the processes are re-mapped following the predefined reconfiguration policy, and the distributed application is restarted. A faulty processor is replaced by a spare processor as long as spare components are available. A possible communication pattern after reconfiguration is shown in Fig. 7b.

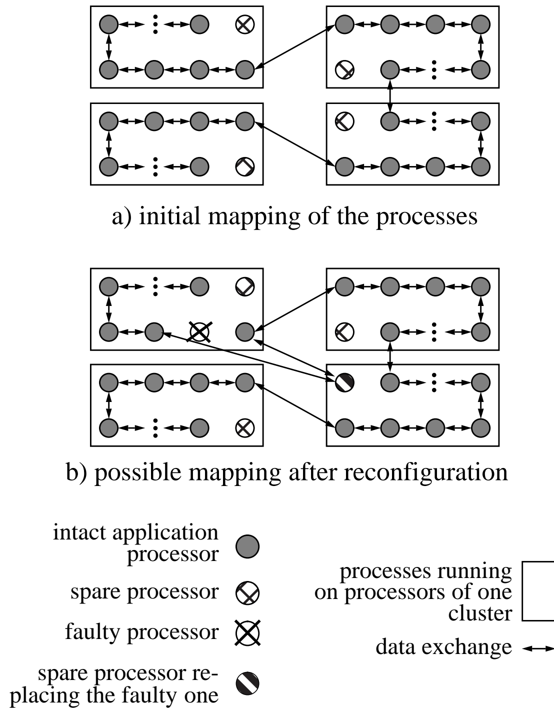


Fig. 7 Linear Array Communication

A single processor failure has no impact on the run time of the algorithm if reconfiguration policy *rec_b*, *rec_d*, or *rec_e* is selected (Fig. 8). This is due to the facts that the spare processor belongs to the same cluster as the faulty processor and within a cluster the interprocessor connections are symmetric and redundant (cf. Fig. 3), i.e., there is no additional communication overhead caused by the reconfiguration. The mean run time of strategies *rec_b, rec_d*, and *rec_e* are clearly shorter than the mean run times of *rec_a* and *rec_c* which show only minor differences for any number of processor failures. *rec_d* requires the largest administration overhead in determining the cluster with smallest distance containing a spare processor but results in the shortest run time. If 8 processors fail in the system with 8 clusters the mean run time deterioration is about 2.2 percent if using the *rec_d* reconfiguration policy and about 3.9 percent for police *rec_a*, which is even worse than the random selection of the spare processor (*rec_c*).

Comparing the ratios of the mean run times of policies *rec_c* and *rec_e* we see the positive impact of the very simple policy in looking for a spare processor in the same cluster. The policy *rec_e* selects a spare processor at random if no spare is available in the cluster of the faulty processor; this strategy shows even better results than the policy to select a spare processor following the internal order of the clusters (*rec_b*). Policy *rec_f* represents the worst case by looking for a spare processor with maximum distance to the faulty processor. Even in the case of a single processor failure, the mean run time is decreased by 3.3 percent, which shows a larger mean communication overhead than in a system with 8 processor failures and reconfiguration policy *rec_b*, *rec_d* or *rec_e*.

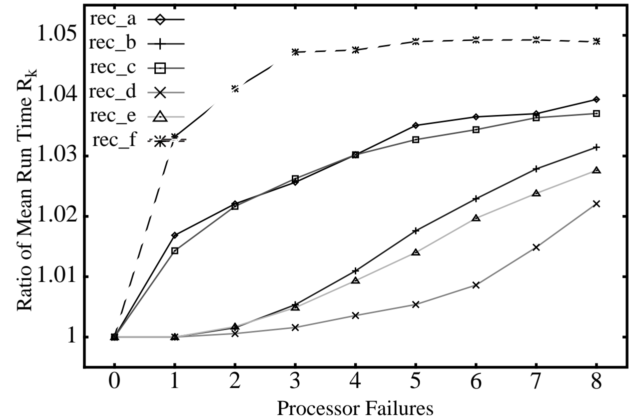


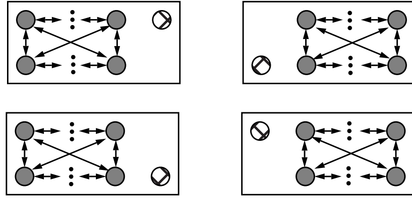
Fig. 8 Linear Array Communication (8 Clusters)

4.2.2. Cluster Internal Communication

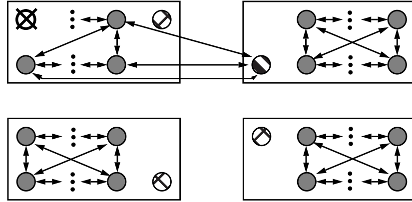
This communication pattern takes into consideration the cluster structure of the multiprocessor. A typical scenario is the assignment of only a single cluster to every user, i.e., a distributed application can be started on the processing nodes of a single cluster but several clusters cannot be allocated by the same user at a time and data exchange between processes initially assigned to different clusters is prohibited.

After every computing phase a cluster internal data exchange takes place. Every process exchanges data with every other of the 16 processes running on processors of the same cluster. Initially, there is no data exchange between processes of different clusters and processes of different clusters can be considered to be independent (Fig. 9a). However, after reconfiguration processes can be mapped on processors of other clusters resulting in intercluster communications (Fig. 9b).

The analyses for *rec_a* and *rec_c* show only minor differences and cause longer mean run times than for the policies *rec_b*, *rec_d,* and *rec_e* (Fig. 10). Already in the case of a single processor failure in a system of 8 clusters these policies result in an average deterioration of more than 6.3 percent. The mean run times of policy *rec_b* are slightly larger than the ones of policy *rec_e* for 3 to 8 processor failures. Policy *rec_f* causes an increase of the mean run time of 15.4 percent if only a single processor fails; this value is even worse than the mean run times of all the other policies if 8 processors fail. The maximum relative loss of performance - in the case of 8 processor failures - is in the range of 8.8 percent (*rec_d*) and 18.2 percent (*rec_f*).



a) initial mapping of the processes



b) possible mapping after reconfiguration

Fig. 9 Cluster Internal Communication

The ratios of mean run times measured in this cluster internal communication pattern are clearly larger than in the previous example (Subsection 4.2.1.). The communication requirements are more intensive; every process exchanges messages with 15 other processes. If a faulty processor is replaced by a spare processor of another cluster, a lot of time-consuming intercluster communications has to be executed causing large deterioration of the overall run time.

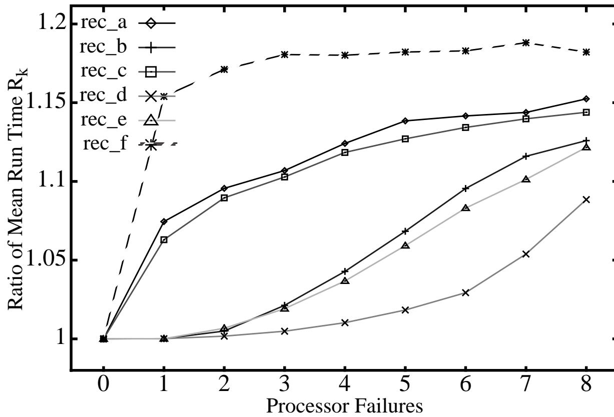
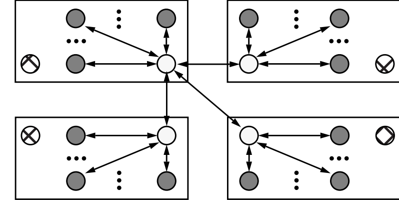


Fig. 10 Cluster Internal Communication (8 Clusters)

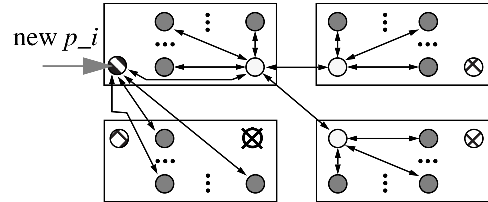
4.2.3. Cluster Oriented Communication

The cluster structure of the target system is considered by this communication pattern in order to reduce intercluster communications in the initial fault-free state.

A central process *p_0* broadcasts the initial data to the other application processes in a tree like manner. In order to reduce intercluster communication, data are sent to one specific process *p_i* per cluster which broadcasts the data



a) initial mapping of the processes



b) possible mapping after reconfiguration

Fig. 11 Cluster Oriented Communication

within the cluster. After the computation phase, *p_i* gathers the data of the other 15 processes of the same cluster and sends a message to the central process *p_0* (Fig. 11a). Several computation and message exchange phases are executed. After a reconfiguration of the system this cluster oriented communication pattern is possibly destroyed (Fig. 11b). If a faulty processor is replaced by a spare processor of another cluster the number of intercluster communications is increased.

When running the application program with a cluster oriented communication pattern, we obtain results similar to the previous communication patterns, i.e., the differences between the ratios of the mean run times of the reconfiguration policies *rec_a* or *rec_c* on the one side, and *rec_b* or *rec_d* on the other side are obvious (Fig. 12). Simulating a system of 8 clusters and assuming 8 processor failures, the mean run time is prolonged by a factor between 1.3 percent (*rec_d*) and 4.75 percent (*rec_f*). It is worth mentioning that reconfiguration policy *rec_d* exerts almost no influence on the mean run time if at most 6 processors fail (deterioration of less than 0.23 percent). Furthermore, policy *rec_e* results in slightly shorter mean run times than policy *rec_d* for 7 and 8 processor failures and delivers better results than policy *rec_b* at any rate.
The results of policy *rec_f* are very different from the previous results of this policy. In this cluster oriented communication pattern the ratios of mean run times increase approximately linearly with the number of processor failures. In the previous examples the mean run time has grown very fast even if only one processor failed and did not increase so clearly if further processors failed.

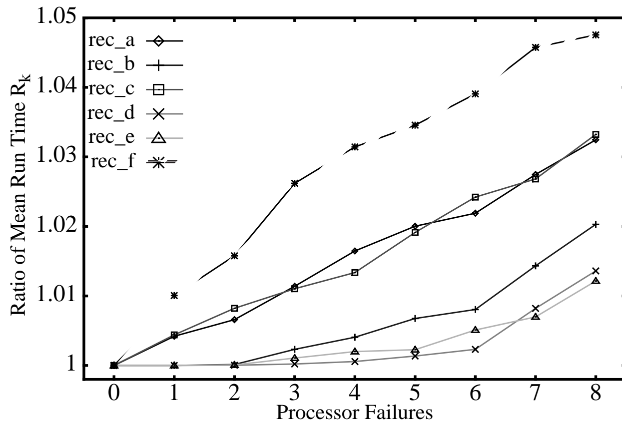


Fig. 12 Cluster Oriented Communication (8 Clusters)

4.2.4. Experiment Summary

The results of the presented simulation experiments considering three different communication patterns and six reconfiguration policies differ in terms of absolute values. The mean deterioration of run times is more obvious in the application with heavy cluster internal communication (Subsection 4.2.2.) than in the other cases. Nevertheless, in all of the considered examples it has been proven that an intelligent reconfiguration policy, which takes into account the hardware architecture of the underlying multiprocessor as well as the communication paths between processors, results in clearly shorter run times for distributed application programs.

5. Summary

Two case studies of combined performance and dependability analysis of fault-tolerant multiprocessors have been carried out. The presented simulation technique has successfully been used allowing detailed evaluation of the target systems including the underlying architecture, fault-tolerance mechanisms, and actual workload. Intensive research is going on to extend the simulation environment and to run more detailed evaluation experiments taking into account various types of multiprocessors, other typical communication patterns and message sizes, different fault-tolerance mechanisms such as fault diagnosis algorithms as well as various workload scenarios. Currently, the components of the MEMSY model MP_M are being refined in order to reflect the internal structure and the communication protocols in more detail. Furthermore, simulation components of I/O subsystems are being added to *SimPar* in order to examine I/O behavior of parallel systems and methods are implemented facilitating easy hierarchical model development of multiprocessor architectures.

Acknowledgments

The authors want to thank the Deutsche Forschungsgemeinschaft (DFG) for supporting the development of MEMSY as part of the Collaborative Research Center SFB 182 and the European Union (EU) for supporting the investigations on the GC architecture in the frame of the ESPRIT project 6731 FTMPS.

References

[1] Dal Cin, M., W. Hohl, J. Hönig and A. Pataricza. *MEMSY - A Modular Expandable Multiprocessor System with Fault Tolerance.* Proc. Parallel Systems Fair of the 8th Intl. Parallel Processing Symposium, Cancun, April 26-29, 1994, IPDS'94 Publication 1994, pp. 21-28.

[2] Dal Cin, M., W. Hohl et al. *Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY*. Proc. First Intl. Conf. on Massively Parallel Computing Systems (MPCS'94), Ischia, May 2-6, 1994, IEEE CS Press, 1994, pp 7-15.

[3] Goswami, Kumar K. and Ravi K. Iyer. *DEPEND: A Simulation-Based Environment for System-Level Dependability Analysis*. Center for Reliable and High-Performance Computing (CRHC), Univ. of Illinois at Urbana-Champaign, 1992.

[4] Goswami, Kumar K. *Design for Dependability: A Simulation-Based Approach*. Ph.D. Thesis, Center for Reliable and High-Performance Computing (CRHC), Univ. of Illinois at Urbana-Champaign, 1993.

[5] Hein, Axel. *SimPar$_{GC}$ - Ein Simulator zur Leistungs- und Zuverlässigkeits-Analyse des Multiprozessorsystems Parsytec GC, Version 1.0.* Internal Report 2/94, IMMD III, Univ. of Erlangen-Nürnberg, 1994.

[6] Hein, Axel. *Evaluation Report on Modelling Tools*. FT-MPS, Esprit Project 6731, Report R 4.2.1; Internal Report 3/94, IMMD III, Univ. of Erlangen-Nürnberg, 1994.

[7] Kreutzer, Wolfgang. *Object Oriented Modelling & Simulation Towards Reusable Frameworks for Simulator Design.* Proc. European Simulation Symposium ESS 94, Istanbul, Turkey, October 9 - 12, 1994.

[8] Meyer, Bertrand. *Object-Oriented Software Construction*. New York [et. al.]: Prentice Hall Internat., 1988.

[9] Parsytec Computer GmbH. *The Parsytec GC Technical Summary*, Version 1.0. Aachen (Germany), 1991.

[10] Schwetman, Herb. *CSIM: A C-Based, Process-Oriented Simulation Language.* Proc. 1986 Winter Simulation Conference (WSC' 86), Washington, D.C., 1986.

[11] Trivedi, Kishor S. and Manish Malhotra. *Reliability and Performability Techniques and Tools: A Survey.* in B. Walke and O. Spaniol (eds), *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, 7. ITG/GI Fachtagung, Aachen, September 1993. Berlin, Heidelberg: Springer-Verlag, 1993.