FTMPS, Esprit Project

Report 2.3.9

On the Realisation of a Fault Tolerance Concept for Massively Parallel Systems

Frank Balbach (ed.), Jörn Altmann, Bernd Bieker, Joao Carreira, Diamantino Costa, Geert Deconinck, Alan Grigg, Axel Hein, Johan Vounckx, Gunda Wenkebach

Institute for Computer Science
IMMD III
Friedrich-Alexander-University Erlangen-Nürnberg
Martensstr. 3
D-91058 Erlangen
http://faui30t.informatik.uni-erlangen.de:1200/

Contents

1. Introduction **1**

2. A Hardware Independent Approach **1**

2.1 The Unifying System Model 1
2.2 The Hardware Independence Layer 2

3. The Hardware Platform **3**

4. Cooperating for Fault Tolerance **5**

4.1 Error Detection 6
4.2 System Level Diagnosis 8
4.2.1 Diagnosis of application processors 8
4.2.2 Diagnosis of control processors 9
4.3 Reconfiguration 10
4.4 Checkpointing and Rollback 11
4.4.1 The distributed controller and the Recovery-line Management 12
4.4.2 The user-triggered or semi-automatic approach 12
4.4.3 The hybrid approach 13
4.5 Operator Site Software 15

5. Validation **16**

5.1 The Modeling Approach 17
5.2 Fault Injectors 18

6. Conclusions **19**

References **i**

1. Introduction

Most users of MPS (Massively Parallel Systems) do not require that these systems are always running with their full capabilities. As long as the system is usable, producing *correct results* and getting performance back to *full* speed within an *acceptable time*, they allow reduced system performance, due to failures of some components or due to down-times required for preventive maintenance.

This need for correctness, continuity, and availability motivates the fault tolerance techniques for long-running number-crunching applications developed within ESPRIT-project 6731. The acronym FTMPS - A Practical Approach to Fault Tolerance in Massively Parallel Systems - explains its primary objective: developing fault tolerance concepts to manage failures in massively parallel computers which are immediately applicable to the market. This implies that the implemented techniques should not require extensive and expensive redundancy in hard- and software and they should be portable.

Within this project four universities and two industrial partners (Parsytec Computer, Aachen/Germany, British Aerospace (MA) Ltd., UK) developed and implemented techniques which cover a wide range of software-based fault tolerance mechanisms: from error detection (Universidade de Coimbra, Portugal) and diagnosis (University of Erlangen-Nürnberg, Germany) to statistics (Medical University of Lübeck, Germany) and recovery mechanisms like reconfiguration, checkpointing and rollback (Katholieke Universiteit Leuven, Belgium and Medical University of Lübeck, Germany).

2. A Hardware Independent Approach

In order to ensure that the mechanisms for fault tolerance are applicable to a wide range of massively parallel systems, a general model of the system architecture together with a hardware independent software interface has been developed.

2.1 The Unifying System Model

The techniques that are developed in the FTMPS project, are generally applicable for the large class of MPS with an hardware architecture that can be represented in the Unifying System Model (USM) [13], [25].

According to the USM, the MPS is (logically) divided in two parts. The data-net (D-net), existing of data-processors, executes the application. The control-net (C-net), existing of control-processors, is responsible for the execution of the system software (downloading of applications, fault tolerance tasks, etc.).

The data-net is further divided in Reconfiguration Entities, which are the smallest units that will be used to (re)configure the MPS. A Reconfiguration Entity consists of a number of active processors, and (possibly) some spare processors. These spare processors allow to replace a failed processor without changing the regular structure of the MPS.

The control-net contains control processors that are shared among multiple applications, and others that are used for a single application. The former define the global control-net; the latter the local control-net. Note that these D- and C-net processors are logically different; on some architectures however, they map to the same physical processors.

A control processor on the local C-net defines a local control entity among the D-net processors. This control entity contains the data-processors that are considered as being failed, when the involved control processor is faulty. Analogously, a global control processor defines a global control entity. In this project, implemented on machines that are conform to the USM, we further assume that the MPS is 2- or 3-dimensional mesh, which is divided in partitions. Every partition is devoted to a single application or user. (Hence space-sharing, no time-sharing.) The MPS is connected via hostlinks to (possibly several) host machines.

Examples of machines that fit in this USM-approach are Parsytec's GC machine (which would have been built with T9000 transputers and C104 routing switches), the Parsytec GCel and the Parsytec (Power)Xplorer series. The Connection Machine CM-5 also provides an explicit control-net. Also the Intel Paragon XP/S fits in this USM approach.

2.2 The Hardware Independence Layer

The Hardware Independence Layer (HIL) provides a standard, functional interface between the various FTMPS software components and the target hardware platform. The code contained in the HIL itself is comprised of all hardware-dependent elements of the higher-level FTMPS software components. This facilitates abstraction of the FTMPS software form the implementation details of the target hardware, enabling such software to be more easily ported onto new hardware platforms. It is therefore simpler for the system designer/integrator to provide system software on a wider range of target hardware as well as exploiting future upgrades in hardware components.

The main disadvantage of this approach is that a system containing a HIL is likely to be less efficient than one which is its functional/hardware equivalent but where the software has been developed with detailed knowledge of the target hardware. However, the potential time, effort and cost benefits provided by adopting this approach are considerable.

Some serious thought and a little investment up-front are required to get a good HIL specification which is stable across many types of processor and which will incur acceptable run-time inefficiencies only. The FTMPS HIL contains a range of system software support services, divided into the following categories:

- system interrogation
- process control.
- communications / routing.
- timing.
- semaphore control.
- memory management.
- host interaction.
- fault detection and checkpointing/recovery.
- other, miscellaneous services.

3. The Hardware Platform

The suitability of the concepts of fault tolerance for massively parallel systems developed within the FTMPS project has been proven by implementations for several systems: the Parsytec PowerXplorer, the Parsytec GCel, and the Parsytec GC/PowerPlus. These computers incorporate all features of a massively parallel system, like scalability, regular system structure, and a large number of processing nodes. Scalability is achieved by extending the hardware in units of 4 processing nodes, for the PowerXplorer, and 16 processing nodes for the GC-series. The PowerXplorer can contain up to 64 processors, the GCel up to 16,384 processors and the GC/PowerPlus up to 2,048 processors. In the following, the main characteristics of the three systems will be described briefly.

The basic unit of the Parsytec PowerXplorer are clusters of 4 processing nodes. Each node is equipped with a PowerPC 601 processor for running the application and an Inmos T805 Transputer, which is used for communication with the neighbouring processing nodes. The 4 hardware links of the T805 are used to build up a 2-dimensional grid topology of the overall system. The system can be physically divided into partitions, where one processing node of each partition has to be connected to a host (see Fig. 1) which is responsible for file access, remote procedure calls, etc.

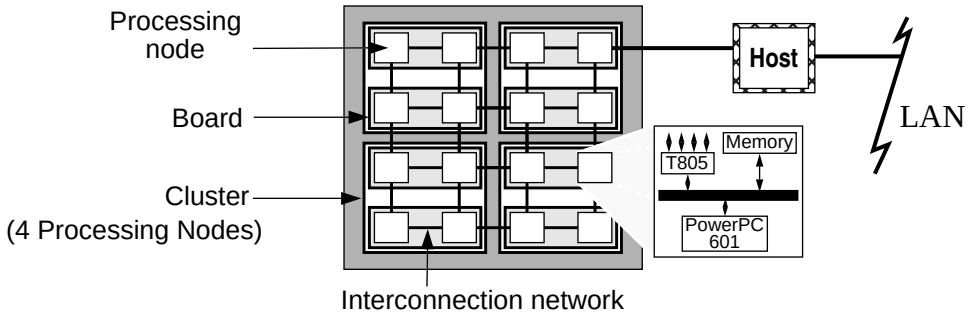

Fig. 1 Structure of the Parsytec PowerXplorer

As there is no physical control-net for PowerXplorer systems, the logical processors of the USM's global and local control-net are physically identical with the data-net processors.

The basic building blocks of the Parsytec GC-systems (GCel and GC/PowerPlus) are clusters of nxn processing nodes (where n=4 for the GCel and n=2 for GC/PowerPlus). The major difference towards the static hardware structure of the PowerXplorer is the possibility of a dynamic configuration of the partitions. This is achieved by the usage of a physically separated control-net (CNet). Each control node can configure the data-net via programmable Inmos C004 switches, resulting in partitions with a two-dimensional grid topology. The nodes of the control-net are connected in a pipeline-structure. 4 clusters of the data-net together with the supervising control node are called a cube.

Whereas the nodes of the Parsytec GCel are built of one T805 Transputer together with local memory, two PowerPC 601 and four T805 Transputers are used for each node of the GC/PowerPlus. Therefore, the task of communication has been separated from the resources needed for executing the application (see Fig. 2).

The physical control-net of Parsytec GC-systems corresponds to the global control-net of the unifying system model (USM). All control nodes used to build up a current partition (i.e. a Reconfiguration Entity of the USM) of a GC's data-net define the local control-net of the USM.

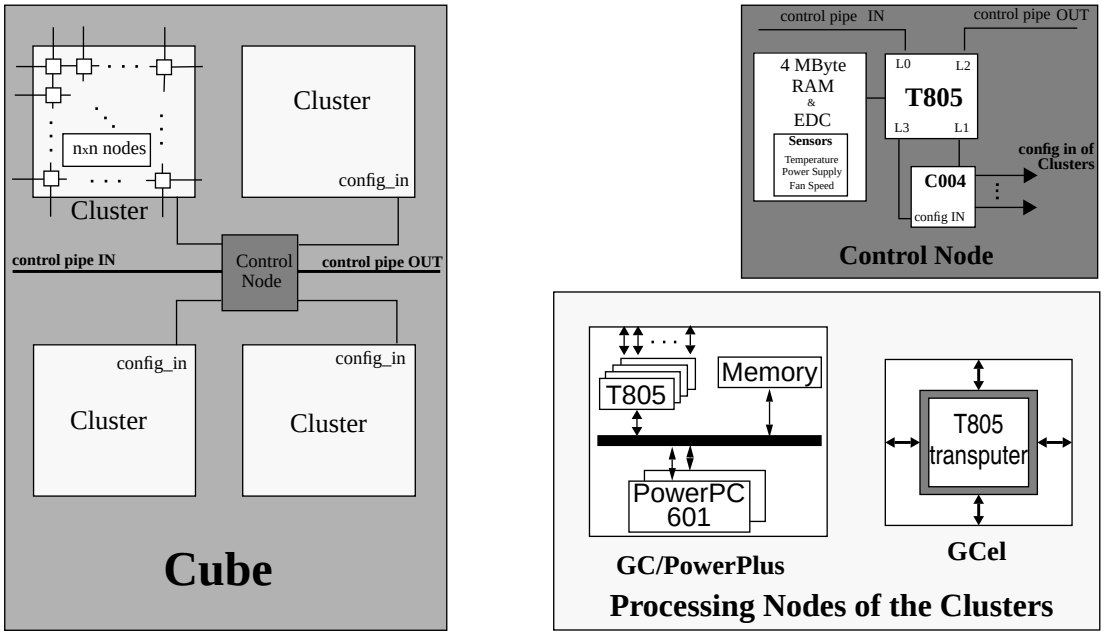

Fig. 2 Structure of Parsytec GC-systems (GC/PowerPlus and GCel)

4. Cooperating for Fault Tolerance

The concept of fault tolerance for massively parallel systems is based on the following functions: concurrent error detection based on *'I am alive'* messages and on-chip error detection; system-level fault diagnosis which analyses and classifies detected errors; user transparent reconfiguration, if the diagnosed fault is permanent or intermittent; restart of failed applications (rollback recovery); and checkpointing (user-triggered and hybrid). These functions are accomplished by the cooperation of several fault tolerance software modules. Fig. 3 shows the cooperating main modules running on the host and on each processor of the MPS.

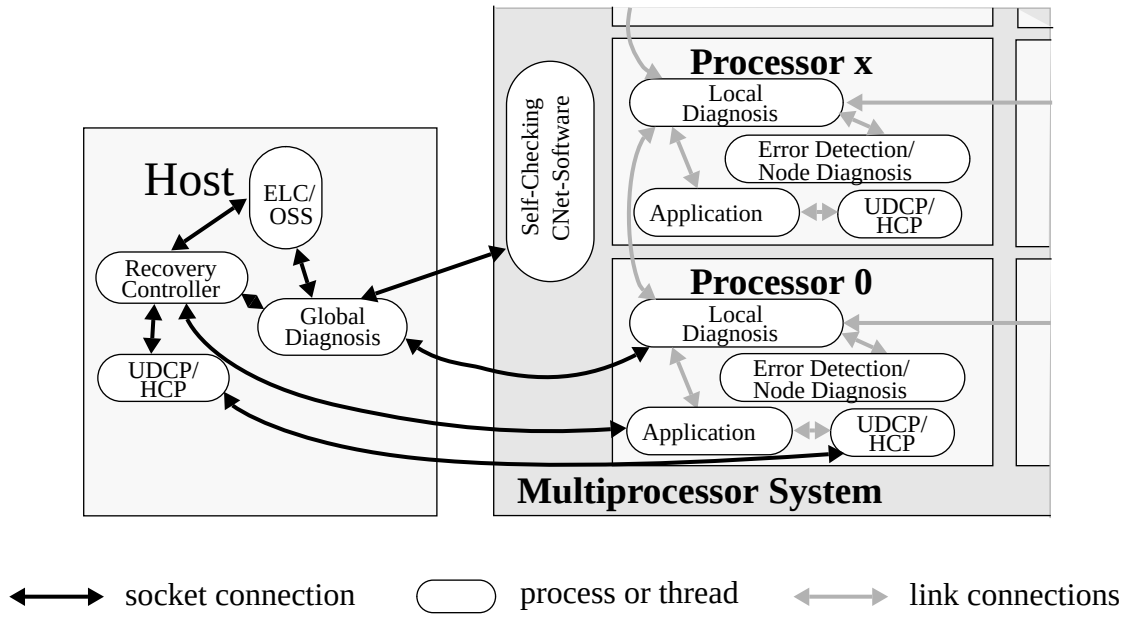

Fig. 3 Structure of the FTMPS-software running on a partition

The **Recovery-Controller** consists of two parts. When a component has failed permanently, the routing tables must be modified such that failed entities are detoured (rerouting). Secondly, when processors fail they must be remapped on spare processors to allow the application to continue (remapping). For saving checkpoints of applications two different approaches were developed. The first one, the **User-Driven-Checkpointing** (**UDCP**), is an economic approach which saves the minimal amount of information at the cost of a limited user-involvement. An alternative to UDCP is the **Hybrid-Checkpointing (HPC)**. This approach to checkpointing releases the application programmer of the burden to specify the data item to be checkpointed. That fault handling by rerouting and remapping is triggered by the **Global-Diagnosis**. It receives the diagnosis results from the **Self-Checking CNet-Software** or from the **Local-Diagnosis** which is a distributed, event-driven, system-level diagnosis algorithm running on the multiprocessor. Detailed information of the error the Local-Diagnosis obtains from the **Error-Detection/Node-Diagnosis** mechanisms for the

application nodes via a link connection. Further essential statistical information about applications and old error occurrences the Global-Diagnosis and the Recovery-Controller are obtained from the **Error-Log-Controller** (**ELC**) which is a part of the **Operator-Site-Software** (**OSS**). Additionally, the OSS covers a set of tasks as the logging and visualization of system status information, and the evaluation of long-term fault statistics.

In the following subsections closer descriptions of these modules are given.

4.1 Error Detection

In FTMPS, from the onset, only hardware faults were considered. Of those, all kinds were taken into account: transient, intermittent and permanent, both in the nodes themselves and in the communication network. An exception was the host - achieving full fault tolerant hosts was outside the scope of the project.

A number of restrictions were also imposed from the start on the kind of error detection methods to use:

- they should not require major changes in the system architecture
- they should not cause (significant) performance degradation
- they should be low cost techniques

The problem was to try to achieve the best possible detection capability under these constraints.

Transient physical faults are one major cause for computer failures. Even in optimal environments it is estimated that transient faults occur 10 to 30 times more frequently than permanent faults, which means that more than 90% of computer failures are caused by transient faults. Furthermore, the decreasing size of chips and the high speed achieved today make them more susceptible to transient faults because the energy difference between logic levels is lower and the timing margins are reduced. Although permanent hardware faults can be detected by periodic or on demand self-test mechanisms, the errors caused by transient faults can only be detected by concurrent error detection techniques. In this way, most error detection methods chosen in FTMPS provide continuous and concurrent error detection in order to detect both transient and permanent errors.

The only error detection methods that are concurrent yet low cost are those based on the monitoring of the behaviour of the system. The traditional technique of duplication and comparison is far too expensive. In the behaviour-based approach, information describing a particular trait of the system behaviour (e.g. the program control flow) is previously collected (normally, this is accomplished at compile/assembly time). At runtime this information is compared with the actual behaviour information gathered from the object system in order to detect deviations from its correct behaviour, i.e. errors. Other examples of behaviour traits are memory access behaviour, hardware control signal behaviour, reasonableness of results,

processor instruction set usage and timing features.

The error detection methods (EDMs) used in the project's last prototype can be grouped in six distinct categories: built-in EDMs, memory access behaviour based EDMs, control-flow monitoring, application level error detection, node level error detection and communication level error detection. Results of the evaluation of these error detection methods can be found in [6],[7].

Built-in EDMs

This mechanisms can be found in the original commercial Parsytec system. They include processor execution model violations (Floating Point exceptions, Illegal instructions, Illegal Operations, Illegal privileged instruction use, I/O segment error, alignment exception), NanoKernel assertions and Parix-level assertions. This mechanisms do not represent any overhead for the applications.

Memory access behaviour

This class of EDMs includes the methods that detect deviations of the proper memory access behaviour (either for instruction fetch or data load and store), namely Accesses to Unused Memory and Writes to the Code Segment. This mechanisms also do not represent any overhead for the applications.

Control flow monitoring

Deviations from the correct control flow are detected by two specific mechanisms: Assigned Signature Monitoring (ASM) and Error Capturing Instructions (ECI). With ASM the program code is divided by the compiler or a post-processor in several blocks, and to each block is assigned a block ID (signature) that does not depend on the block instructions. Whenever a block is entered, that ID is stored in a fixed place, and when a block is left a verification is made that its ID is stored at that place. Due to the need to add, to the application, code to perform that storage and verification, this method has some performance and memory overhead. With ECI, trap instructions are inserted in places where they should never be executed (for instance, after an unconditional jump). Only if something goes wrong will one of them be executed, thus detecting an error.

Application level EDMs

Contrary to the previous methods, the following methods are visible to the programmer and managed by him. The first one is the application level watchdog timer, that monitors the system's behaviour in the time domain by establishing, for each part of the computation, a time-out that can only be exceeded in the case of a fault. The second one consists of application-level assertions - invariants that can be verified by the application independently of the data that is being processed. For instance, in the Ising test program, a verification is made at several points to see that the spin values are indeed either 1 or -1. This method is totally dependent on the programmer.

Node level watchdog timer ("I'm alive" mechanism)

The “I’m alive” mechanism was implemented as a part of the System Diagnosis layer, and consists of processes that periodically send messages to the processor’s neigbours, to verify that all of them are still alive.

Communication level error detection

At the communication level, only a single mechanism was implemented: the CRC verification of the integrity of the messages.

4.2 System Level Diagnosis

The diagnosis algorithm is comprised of two parts: one executed on the multiprocessor (on the control net and on the data-net) and the other one executed on the host. Due to the partitioning of multiprocessors it is necessary to start the local-diagnosis always together with an application within a partition. To handle this situation three kinds of diagnosis processes running on the host were introduced. The main process works as a daemon and is called the **Global-Diagnosis**. This process is started only once when the system is booted and exchanges information with the Error-Log-Controller (ELC) and the Recovery-Controller. Furthermore, for each application running on the multiprocessor a process is started which is called **Partition-Wide-Diagnosis**. It manages the communication to the **Local-Diagnosis** on the multiprocessor and starts a process called **Test-Host-Link** which tests the link connection to the partition of the multiprocessor executing the application. Additionally, the **Global-Diagnosis** has a connection to the **Self-Checking-CNet-Software** which sends error reports if errors will occur on the control net.

The part of the diagnosis algorithm running on the multiprocessor is described in the following two subsections: diagnosis of application processors (**Local-Diagnosis**) and diagnosis of control processors (**Self-Checking-CNet-Software**).

4.2.1 Diagnosis of application processors

The part of the distributed diagnosis algorithm diagnosing the application processors (Local-Diagnosis) integrates error detection mechanisms and minimizes the number of diagnostic messages. The aim of the algorithm is to generate a correct diagnostic image in every fault-free processor. If the diagnosis is correct, the fault-free processors can logically disconnect the faulty units from the system by stopping the communication with them. Employing this method, the number of tolerable faults depends only on the properties of the system interconnection topology. Furthermore, a new syndrome decoding method is used which produces the diagnosis gradually.

The system-level diagnosis algorithm is distributed, which makes it applicable in scalable systems; and event-driven, thus it processes diagnostic information fast and efficiently, requiring small amount of communication and computation [2].

The main structure of the implementation is shown in *Fig. 4.* If no fault event is detected, the algorithm periodically tests the neighboring processors. Testing is accomplished by assigning independent modules to each tested unit.

If the tests detect an error in one of the neighboring processors, exception handling is invoked by issuing an error indication from the corresponding *testing* module to the *local diagnosis* module. The local diagnosis module gives control to the *supervisor* module, which handles the exceptions caused by the detected error. The supervisor module activates the modules responsible for *terminating the current application*, for *distribution of the local test results*, and for *processing of the diagnostic information*. The algorithm executes the local test result distribution and the syndrome decoding procedures alternatively, thus creating diagnostic images gradually, taking every test outcome into consideration during diagnosis.

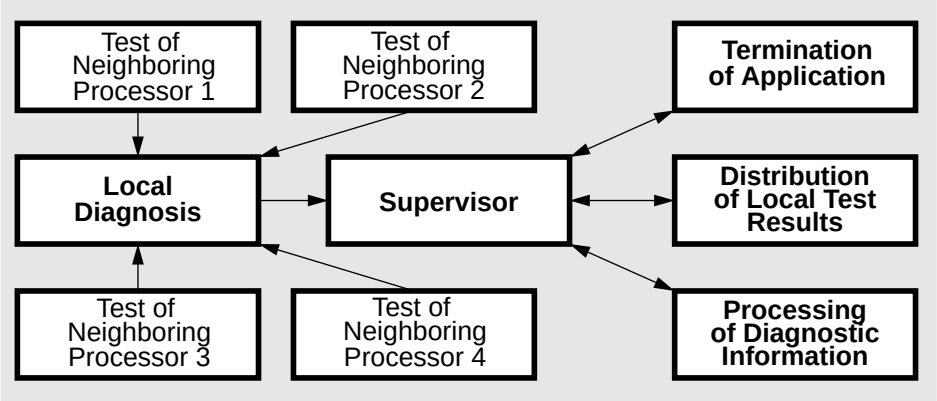

Fig. 4. Main modules of the implemented algorithm

The implementation was examined, highlighting the advantages and disadvantages, in [3]. Furthermore, we have proven the efficiency of our algorithm by some measurement results. The measurement results show that the testing can cause only a small overhead (less than 0.5% if the “I’m alive” messages are sent each 1.0 seconds).

4.2.2 Diagnosis of control processors

Mechanisms for fault tolerance have to be included if the hardware platform is equipped with an additional control network, i.e., the CNet in systems of the Parsytec GC-series. Therefore, a modular concept for fault tolerance extensions to the CNet-Software has been developed, which can be reused for future versions of the CNet-Software as well as for other Parix-applications. This has been achieved by only having a minimal interface to the fault tolerance extensions, i.e., by just calling one function and recompiling the sources of the CNet software. It is called the Self-Checking-Cnet-Software.

In order to be able to detect errors within the CNet, an error detecting router based on top of the Parix-router has been developed. By sending all messages of the CNet-Software using

this router, errors during communication are detected by checking a generated CRC of the messages. Crashed processors of the CNet are detected by the absence of <I'm alive> messages, which are sent between neighboring processors and between the host and the processor connected to the host. Control flow errors are detected by instructions for checking the correct control flow. The instructions for extending the source code are generated automatically using a preprocessor.

As soon as an error is detected, the CNet-Software and all applications will be stopped. The classification of the error (permanent or transient) will then be done by loading hardware test programs on the CNet which have been developed using the OCCAM-programing language. The types of faults which are covered by the tests implemented are shown in Table 1:

Table 1: Types of faults

Memory faults	Processor faults	EDC-Logic faults	Link faults
Stuck-at-faults	Decoding of registers	Stuck-At-faults	Connection
Faults in address-logic	ALU	No Correction	To/From C004
Transition faults	FPU	Wrong Detection	To/From T805
Loss of data			

4.3 Reconfiguration

If a permanent failure occurs, the system must be reconfigured before the application can be restarted. The reconfiguration strategy [17] must provide each (affected) application with a partition on which the application can resume operation. This requires that the partition contains sufficient working processors and that all of the working processors must be able to communicate with each other. The reconfiguration problem can be divided into several subtasks:

- Fault isolation at partition level is obtained by a double blocking mechanism. The (re)configuration algorithm must provide this blocking when setting up the partition borders. Only if the nodes at both sides of the border are faulty, a message can cross partition boundaries. This is perfectly acceptable however, since both partition then contain faulty components and must be restarted anyway.

- Each application needs a partition containing a minimal number of working processors. After a failure this is no longer guaranteed. Hence, the system must possibly be repartitioned [18]. The repartitioning algorithm must provide each affected application with a new / extended partition containing enough working processors. Since we work with massively parallel computers, the complexity of this algorithm is of crucial importance. Hence we developed an algorithm with complexity polynomially proportional to the number of allocated partitions, rather than to the number of processors in

the system.

- The partition must be booted again. Since the network is injured a special loader is necessary [19]. Again emphasis is put on the complexity. The execution time complexity is kept proportional to the diameter of the boot network. The data complexity is proportional to the number of faults in the partition. Once the partition is booted, the run-time kernel (with fault tolerance extensions) can be activated.

- An important aspect of the run-time kernel are its routing functions. A fault tolerant routing algorithm must route messages between any two working nodes of the partition. Classical routing tables using a look-up table have a data complexity proportional to the number of processors in the partition. In massively parallel computers this is no longer feasible. Hence we developed a fault tolerant routing algorithm with a compact representation of the routing information. This compact representation is based on interval routing [16], [24].
  The fault tolerant routing algorithm is based upon rerouting the messages along faulty components. The regularity of the original grid topology is extensively used to reduce the complexity of the routing information. We developed several compact fault tolerant routing algorithms [20][21][22][23], of which the system operator or the user can choose, depending on the application's requirements and the system. The developed algorithms are valid for 2-dimensional grids and more generally n-dimensional grids. A minimal (shortest path) version is available.

- The application must be shielded from the fault tolerance measures. It should see a (virtually) perfect system. This implies that we need to map this virtually perfect system on an injured one. The remapping algorithm will take care of that by assigning each logical processor to a physical one.
  The remapping follows a two-way approach. At first, we try to remap the processors locally if enough spare processors are available in the reconfiguration entities. This graph-theoretical approach offers the advantage of preserving the system's properties. If not enough spare processing power is available, the application topology is embedded on the injured physical graph. Since less system resources are available, the overall performance of the system will decrease (graceful degadation).

4.4 Checkpointing and Rollback

The application recovery in the FTMPS project is based on coordinated (consistent) checkpointing [4], [10], [12], [25]. This means that periodically a consistent view of the application is saved onto secondary storage (checkpointing); after a fault occurred, the application is then restarted from the most recent valid set of checkpoints (recovery-line). The implementation consists of two parts. The first part is a distributed controller which executes the recovery-line management. The second part triggers the checkpointing and rollback in a

data-driven approach. The controller part is common to each of these interfaces.

4.4.1 The distributed controller and the Recovery-line Management

The distributed controller is implemented for a distributed disk system and consists of several sets of processes (one set per disk). On the nodes connected to the disks, the controllers are started automatically via a command-line option when the application is started. These controllers manage the checkpoint data and are (logically) connected via a scalable, hierarchical tree. They save their state in a small database and exchange control messages to have a consistent view.

The first task of the distributed controller is the recovery-line management. During the normal operation, a controller keeps track of which checkpoint data from which processes are saved. Further, it agrees with the other controllers about which recovery-lines become complete and valid. This consistent recovery-line determination takes the assumption of a non-zero error-detection latency is taken into account (the fail-time-bounded fault model). It assumes that problems are reported within a (user-definable) period, after which a recovery-line is declared valid and may be used to roll back to. Upon rollback, the controllers determine which valid recovery-line is the most recent one, such that the corresponding checkpoint data can be restored in the application processes. The fail-time-bounded fault model implies that failures during recovery do not endanger the consistency. The second task of the controllers, the garbage collection, cleans up the space on the disks, occupied by old, corrupt or incomplete recovery-lines.

This controller part is common for each of the following approaches.

4.4.2 The user-triggered or semi-automatic approach

The second part of the application recovery is the interface to the application. In the UDCP-tool (user-driven checkpointing tool), which implements the semi-automatic approach, a library is linked to application, such that the programmer can trigger the checkpointing and rollback at run-time. The library (C or FORTRAN) consists of a set of calls to specify the contents of the checkpoint and a call to trigger the checkpointing.

The contents of the checkpoint is specified by defining which data-items of the process contribute to the consistent state of the application at the recovery-line. In C, a generic routine allows to incorporate the data area of this item in the contents of the checkpoint (e.g. variables, arrays, structures, pointers, etc.). Analogously, FORTRAN-calls are provided to put arrays of integer, real, complex, ... data in the contents of the checkpoint. Important is also that files can be included in the contents of the checkpoint (in contrast to many other checkpointing approaches that require closed systems without file I/O). These files may be shared among several processes (both with read-only and read/write access) [11].

The recovery-line (which shows where a global consistent state can be found in every pro-

cess of the parallel application) is indicated by a single call. This will trigger the checkpointing in the application at run-time. This call will also trigger the restoring of the application's state when it is restarted.

At run-time the following sequence of actions occur. During normal execution, the application does its initialization (e.g., set-up of communication channels, allocation of memory space, etc.). Then, the checkpoint contents is initialized with the data-items, as specified by the programmer. At the moment the recovery-line is reached, the checkpointing is triggered. This means that the states of the specified data-items are sent to the disks, together with some control information (process id., recovery-line number, ...). Then, the application process continues its execution, until the next checkpointing is triggered.

Upon rollback, the distributed controller determines which set of checkpoint data (recovery-line) will be used for this rollback. Then, the application starts its execution and initializes its specific data (e.g. re-building topology). Further, the contents of the checkpoint is initialized with the data-items specified by the user. (This assures that when the program is loaded at another start-address, the values of the variables will be restored on their right places. Hence, this allows address relocation and does not require a deterministic loader.) At the place where the recovery-line is encountered (where the checkpointing was triggered during normal execution), the states of the data-items are restored. This means that the data-items are set to their saved values. From here, the application continues from its normal execution until the following checkpointing is triggered. An optimization allows to skip unnecessary parts upon rollback.

During normal execution, overhead only occurs during the checkpointing. This overhead is proportional to the checkpoint size (which does not contain any inter-process communication data) and to the available I/O bandwidth to the disc. Apart from the initialization of checkpoint contents, and a few control messages for the recovery-line management, there is no overhead for the application between the moments that checkpoints are taken. The non-blocking approach (only for file-checkpointing, a partial synchronization barrier is sometimes necessary [11]) and the small checkpoint size (because only the data-items indicated by the programmer have to be saved, and not intermediate variables) minimize this overhead. Besides, the checkpointing method is hardware independent: the contents of the checkpoint contains only data-items from the application processes and no (system or compiler) dependent data for heap, stack, communication structure, etc. The main effort is the user involvement.

4.4.3 The hybrid approach

This approach is established in order to reduce the effort the user has to provide for achieving a fault-tolerant execution of the application program. In user-driven checkpointing, besides the initialization phase the user has to define two things: the data items to be checkpointed and the position within the source code where the checkpoint storage call is to be executed. The latter one consists only in a function call but it has to be assured that

throughout the partition the application is running on a consistent view is stored. Due to the knowledge on the application this part can be done by the programmer. The definition of the checkpoint content can be quite tedious since many entries might be necessary. In the hybrid checkpointing the initialization is as in the user-driven case. But here, there is no need for defining the data items to be checkpointed. Nevertheless, the checkpoint storage calls have to be executed. The actions to be done for a normal start are:

- redefine the SendLink and RecvLink functions by FT_SendLink and FT_RecvLink, Wait and Signal by FT_Wait, and FT_Signal and malloc / free by FT_malloc / FT_free.
- include a BuildCP_Init() call as the first instruction into the application program.
- at the positions in code where the checkpoint has to be saved insert a BuildCP() call (this has only to be done for one process on every node).
- recompilation of the application and linking of a checkpointing library.
- for the start a runscript is provided enabling the user to select a link, the x- and y-dimension of the partition and to set a flag for NormalStart or Rollback.

When the BuildCP_Init() function is performed, the actual status of the communication system contained in the runtime system is saved. This information comprises of the pointers to the available local and external communication links. Next a initialization sequence similar to the one used for user-driven checkpointing is executed.

Due to the redefinitions described above the dynamic allocation of memory areas can be logged as well as the usage of semaphores. This information is to be included into the checkpoint as well.

Upon execution of the BuildCP function all application processes have to be included into the checkpoint. Therefore, the run queue is checked as well as the communication channels and the semaphores. Each process is stored with its corresponding instruction pointer. This is necessary for a proper restart upon rollback. Beside this, information on the communication links and the semaphores used has to be included as well. Next, all memory areas allocated by the application are included into the checkpoint. This areas consist of the heap, stack and static area of the application and all dynamically allocated memory areas.

In case of a rollback, first all memory areas used by the application will be restored out of the checkpoint. Afterwards, all communication channels and semaphores used by the application will be set to the status saved. At a last step all application processes will be restarted again. So, for the rollback in the hybrid approach it is not necessary to run again through the initialization phase. It is possible to directly start the application from the position in code were the BuildCP call was performed.

For the hybrid-checkpointing some functions of the runtime system had to be replaced. The new functions provided encapsulate the original functions but log the information handed to these functions. Due to this logging a small overhead occurs at every call of one of these functions. For the execution time of the BuildCP function the application processes are

blocked. Since the checkpoints are rather large (the entire heap and stack areas are to be saved) the overhead will be larger than in the user-driven approach. Nevertheless, for complex applications where the definition of relevant data for the checkpoint is difficult this approach offers an easy to use possibility to add fault tolerance to an application.

4.5 Operator Site Software

The interface between the fault tolerance software and the operator (administrator) of a massively parallel system is set-up by the Operator-Site Software (OSS). The OSS covers a set of tasks including the logging and visualization of system status information, the evaluation of long-term fault statistics, and the support of the diagnosis and recovery software with several information about distinct faults. According to these tasks the OSS is modular structured and consists of a Database Tool, a Statistic Tool, a Visualization Tool, and modules for automatic error logging and database access (Error-Log-Controller and Application Controller). Since System Status Visualization was not considered to be important for the project, only a demonstration version of this part of the OSS exists.

The OSS can be divided into three functional blocks:

1) Database
2) on-line part= OSS-Control-Modules
3) off-line part= OSS-Tools

The on-line parts (OSS-Control= Error-Log-Controller and Application Controller) work in close interaction with the other FTMPS software. They are active whenever the fault tolerance is used. The off-line tools (OSS-Tools= OSS Database Tool, OSS Statistic Tool, System Status Visualization) can be opened and closed by the system operator whenever they are needed. The global structure of the OSS is illustrated in Fig. 5.

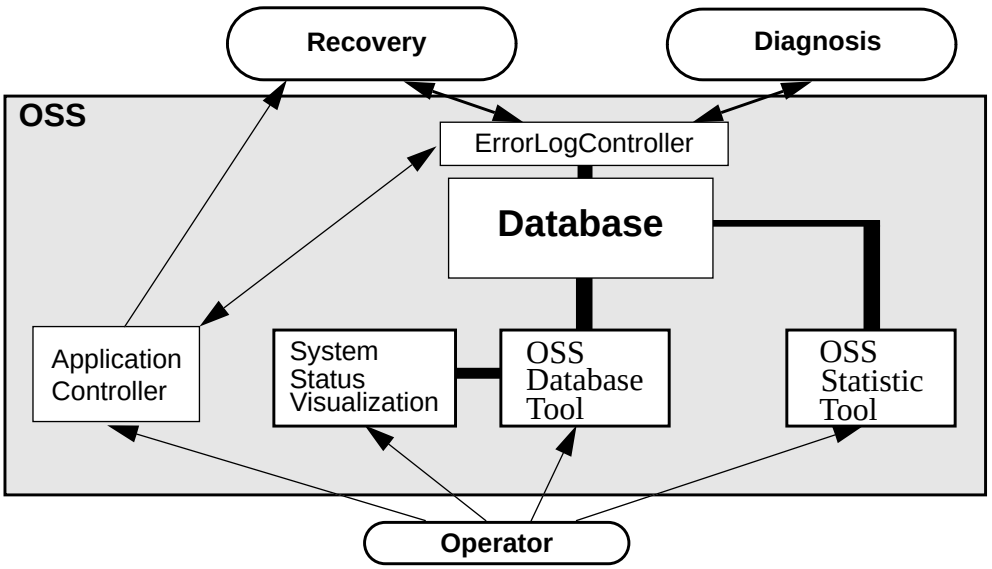

Fig. 5 Global Structure of the Operator-Site Software

The Error-Log-Controller

Automatic logging of hardware faults in the parallel system is done by the Error-Log-Controller (ELC). The ELC runs on the control host of the parallel system and builds the interface between the database relations and the other parts of the FTMPS that need access to the database. There are communication links to the host modules of the Diagnosis and the Recovery software and to a module called Application Controller (AC).

In cooperation with these processes the ELC performs several tasks:

- Logging of fault reports coming from the Diagnosis host module.
- Providing the Diagnosis software with information about distinct faults.
- Providing the Diagnosis and Recovery software with a list of all currently defective components in the system.
- The Recovery informs the ELC when an application is started, stopped, remapped or restarted after a failure. As a result the ELC updates the according database relation
- The result of an operator request is sent by the Recovery to the ELC.
- When the operator uses the OSS-Tools, the ELC is stopped by the AC until the operator is ready. During that time, the ELC can't update the database. That's the reason why a copy of the current database should be used for the OSS-Tool.

The Application Controller

In order to decide if an application should be remapped to another partition of the system after a permanent fault occurred, the recovery controller needs a list of all running applications. This information is provided by the ELC. The list is kept up-to-date by the recovery controller itself (all starts and stops of applications are reported to the ELC) and by the system operator. For the operator a graphical interface is provided - the Application Controller - that gives him access to the application database and that allows him to send requests (start application, stop application, remap application) to the recovery module.

Additionally, the operator can verify the database containing the permanent faults using the Application Controller. This may be necessary after repairs or when components or parts of the system fail and this is not reported by the Diagnosis software. To ensure that inconsistencies are avoided, the Error-Log-Controller is blocked by the Application Controller while the operator accesses the database files

A functional view of the Operator-Site Software is given in the Report R4.3.8/A [26]. dIn the Report R4.3.7 [27] the software-structure of the OSS-Tool ‘System Status Visualization’ is described.

5. Validation

The developed fault tolerance concepts can be validated by various methods such as measurements, analytic modeling or simulation. In each case dependability and performance are

not analyzed separately. A combined performance and dependability analysis - *performability* analysis - of the overall system is conducted.

Simulation models are common evaluation methods during the early design phase and describe the target system as a computer program. Therefore, the models are much more powerful, more flexible, and less restricted than analytical models. Here, we consider only *discrete event simulation* allowing the state of the system to change instantaneously at distinct points in time. Various distribution functions can be included in a model and the model designer is not restricted to the exponential distribution as in a continuous-time Markov process. When large and complex systems need to be analyzed considering lots of details, simulation is the only technique to perform this task in a realistic way.

Measurements based on monitoring are only possible when the real machine or a prototype already exists. To validate our fault tolerance concept by measurements a fault injector for the Parsytec machines has been developed. Obviously, measurements produce the most trustworthy validation results. However, in this late phase of system development the hardware and major parts of the system software are fixed and cannot easily be changed.

5.1 The Modeling Approach

In order to perform a sophisticated analysis of fault-tolerant multiprocessors the system-level simulator *SimPar* has been developed for the performance and dependability evaluation [8], [9],[14],[15]. To implement *SimPar* the *Depend* tool developed at the CRHC (Center for Reliable and High Performance Computing) of the University of Illinois at Urbana-Champaign is taken as the underlying simulation engine which provides basic components such as simulation classes for fault-tolerant servers and TMR systems. It has successfully been used for the analysis of a TMR-based system and for the simulation of software behaviour under hardware faults. *SimPar* provides new enhancements to facilitate the model development and the performability 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 possibility of *fault injection* into components of the simulation environment in order to interrupt and disturb their regular and predefined behaviour.

Faults can be injected in objects of every basic class disturbing and interrupting the predefined fault-free behaviour in order to examine the influence of local fault occurrences on the overall system. The model designer can define its own fault model by assigning functions to the basic classes which are called as soon as faults are injected.

The hierarchical construction of a *SimPar* architecture model may reflect the USM and hierarchical topology of the target architectures. Based on the cluster hierarchy the models can easily be kept scalable. The complicated insertion of the connections between basic components and the cumbersome initialization of the routing tables is simplified and adapt-

ed to varying model sizes. Using the very flexible addressing mode, it is easy to access the basic components or groups of basic components on the different levels of the cluster hierarchy in order to assign workload to selected processor objects, to define local error injections, or to query status informations of basic components.

5.2 Fault Injectors

The growing complexity of both the hardware and software of contemporary computers makes the evaluation of the dependability properties of the systems more and more difficult. This is clearly true for methods based on analytical modelling, but it is also true for experimental methods based on fault injection. Traditional physical fault injection approaches such as pin-level fault injection, heavy-ion radiation, and power supply disturbances are very difficult to apply in contemporary systems. In fact, the high complexity and the very high speed of the processors available today make the design of the special hardware required by the above techniques very difficult, or even impossible. The main problem is not in the injection of the faults itself but it is related to the difficulties of controlling and observing the fault effects inside the processor. Simulation based fault injection also has serious problems in being used in very complex processors, as the development effort (of the fault injection tools) and the time consumed by the experiments dramatically increases with the complexity of the processors. Software fault injection techniques, also known as fault emulation, are being increasingly used as an alternative to the other methods. It basically consists of interrupting the application execution in some way (usually by inserting a software trap or executing the application in trace mode) and executing specific fault injection software code that emulates hardware faults by inserting errors in different parts of the system such as the processor registers, the memory, or the application code. The main advantages of software fault injection are its low complexity, its low development effort, and its slow cost (no specific hardware is needed). This is the approach followed in the FTMPS project. However, in spite of the large number of software fault injection works and the considerable advances proposed in the last years, existing software fault injection tools still have several problems and limitations: - they have considerable impact on the target system behaviour - they have a restricted range of fault triggers - the target system monitoring required either for detecting the activation of some faults or to collect relevant information on the fault impact, is very limited.

In the FTMPS project, a new approach was developed, that consists of the use of the advanced debugging and performance monitoring features existing in modern processors like the MPC601, to inject more realistic faults by software, and to monitor the activation of the faults and their impact on the target system behaviour in detail. A new software fault injection and monitoring environment based on this idea has been built for the environment PARIX/PowerPC, called PowerSFI. A general version of it, called Xception, is described in [5], with some results of its application to the FTMPS prototypes presented in [7].

By directly programming the debugging hardware inside the MPC601, PowerSFI can inject faults with minimum interference with the target application, that is not modified. Ad-

ditionally, a number of sophisticated triggers is available, including fault triggers related with the manipulation of data, and very detailed information about the target processor behaviour after the fault is recorded (like number of clock cycles, the number of memory read and write cycles, and instructions executed).

But, in the FTMPS project, faults had to be injected not only in the processor, the job of PowerSFI, but also in the communication subsystem, a very important part of a parallel machine. This required the development of a totally different fault injector, called CSFI-PPC (Communication Software Fault Injector for the PowerPC). The approach consists basically in modifying the PARIX router software in order to corrupt received/routed packets, providing tools for the user to control and automate the fault injection process. A description of this injector and of some of the obtained results can be found in [6].

6. Conclusions

MPS are typically used for number-crunching applications which are often running for days or even weeks for completion. Hence, the failure of even a part of the system is costly, because the calculations have to be restarted from the beginning. As the statistical occurrence of a fault in a MPS increases with the number of processing elements, fault tolerance is needed to restrict the performance loss. Fault tolerance software can imply that errors are detected, the faults diagnosed and the system and application are recovered. The FTMPS project combines these different steps in a global approach to fault tolerance for massively parallel systems and integrates it in system management software for the operator. Typical features of MPS like scalability or a separated control net has been taken into account by developing the approach for a general model of the system hard- and software.

The built-in hardware error-detection features combined with software error-detection techniques provide a high coverage of transient as well as permanent faults. Combined with the diagnosis software the necessary information for the OSS and the reconfiguration is collected. Backward error recovery is based on checkpointing and rollback. Since checkpointing introduces some overhead concerning the overall execution time different approaches are provided. These approaches require more or less user involvement. Depending on the execution time of the application and the mean-time-between-failures for the target machine the user can choose among user-driven and hybrid- checkpointing. Thus, either more effort has to be put to the user by adding only little overhead to the application`s execution time or little user involvement is required that has to be paid for by a larger overhead.

This global approach to fault tolerance has also been implemented on a commercial MPS. The different modules of the fault tolerance software cooperate, such that, for the user, the system continues to work in spite of errors. Besides, the several modules can be used in stand alone mode. This enables the system operator to select the degree of fault tolerance to the needs of the application or the users.

References

[1] Altmann, J., 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,“ *IEEE 1st European Dependable Computing Conference*, pp. 371-385, Berlin, Germany, October 1994.

[2] Altmann, J., T. Bartha, and A. Pataricza, ”An Event-driven Approach to Multiprocessor Diagnosis,“ *8th Symposium on Microcomputer and Microprocessor Application*, *μP'94*, pp. 109-118, Budapest, Hungary, 12-14th October 1994.

[3] Altmann, J., T. Bartha, and A. Pataricza, ”On Integrating Error Detection into a Fault Diagnosis Algorithm for Massively Parallel Computers,“ *1st International Computer Performance and Dependability Symposium*, *IPDS'95*, pp.154-164, Erlangen, Germany, 24-26th April 1995.

[4] Bieker, B., G. Deconinck, E. Maehle, J. Vounckx, “Reconfiguration and Checkpointing in Massively Parallel Systems”, Proc. of EDCC-1, L.N.C.S. 852, Springer-Verlag, Berlin, Germany, Oct. 1994, pp. 353-370

[5] Carreira, J., H. Madeira, João Gabriel Silva “Xception: Software Fault Injection and Monitoring in Processor Functional Units” Proceedings of the “Fifth IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5), Urbana-Champaign, Illinois, USA, 27-29 September 1995.

[6] Carreira, J., H. Madeira, João Gabriel Silva. “Assessing the Effects of Communication Faults on Parallel Applications” Proceeddings of the 1st International Computer Performance and Dependability Symposium (IPDS'95), Erlangen, Germany, 24-26 April 1995, pp.214-223, IEEE Computer Society Press, ISBN 0-8186-7059-2.

[7] Costa, D., F.Moreira, H.Madeira, M.Rela and João Gabriel Silva, “Experimental Evaluation of the Impact of Processor Faults on Parallel Applications”, Proceeddings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS-14), Bad Neuenahr, Germany, 13-15 September 1995.

[8] Dalibor, S., A. Hein, and W. Hohl., "Simulation-Based Performability Evaluation of Fault-Tolerant Multiprocessors.", in Proceedings of the ESM '95, European Simulation Multiconference, Prague (Czech Republic), June 5-7, 1995.

[9] Dalibor, S., A. Hein, and W. Hohl., "Application Dependent Performability Evaluation of Fault-Tolerant Multiprocessors.", accepted at Euromicro's 4th Workshop on Parallel and Distributed Processing PDP `96, Braga (Portugal), January 24-26, 1996.

[10] Deconinck, G., J. Vounckx, R. Lauwereins, J.A. Peperstraete, “Survey of Backward Error Recovery Techniques for Multicomputers Based on Checkpointing and Rollback”, Proc. IASTED Int. Conf. Modeling and Simulation, Pittsburgh, PA, May 1993, pp. 262-265

[11] Deconinck, G., J. Vounckx, R. Lauwereins, “The Consistent File-Status in a User-Triggered Checkpointing Approach”, ParCo’95, Ghent, Belgium, Sep. 1995

[12] Deconinck, G., J. Vounckx, R. Lauwereins, J.A. Peperstraete “A User-triggered Checkpointing Library for Computation-intensive Applications”, Proceedings Seventh IASTED-ISMM Int. Conf. On Parallel and Distributed Computing and Systems, Washington, DC, Oct. 19 21, 1995.

[13] Deconinck, G., J. Vounckx, R. Cuyvers, R. Lauwereins, B. Bieker, H. Willeke, E. Maehle, A. Hein, F. Balbach, J. Altmann, M. Dal Cin, H. Madeira, J.G. Silva, R. Wagner, G. Viehöver, “Fault-Tolerance in Massively Parallel Systems”, accepted for publication in Transputer Communications.

[14] Hein, A., "A Process-Oriented Simulation Model for the Performability Analysis of a Fault-Tolerant Multiprocessor System",in proceedings of the ESS ‘94 (European Simulation Symposium), Istanbul (Turkey), October 9-12, 1994.

[15] Hein, A., K. Goswami.,"Combined Performance and Dependability Evaluation with “Conjoint Simulation”.in proceedings of the ESS ‘95 (European Simulation Symposium), Erlangen (Germany), October 26-28, 1995.

[16] van Leeuwen, J., R.B. Tan, Interval Routing, The Computer Journal, vol. 30, no. 4, 1987, pp. 298-307

[17] Vounckx, J., G. Deconinck, R. Lauwereins, "Reconfiguration of Massively Parallel Systems", HPCN Europe 95 conference, Milan, Italy, May 3-5 1995, Lecture Notes in Computer Science 919, Springer-Verlag, pp. 372, 377

[18] Vounckx, J., G. Deconinck, R. Lauwereins, "Algorithms and documentation for repartitioning", report D3.1.19, ESPRIT Project 6731 (FTMPS), August 1995

[19] Vounckx, J., G. Deconinck, R. Lauwereins, J.A. Peperstraete, "A Loader for Injured Massively Parallel Networks", PDCS, October 1995, Washington, USA

[20] Vounckx, J., G. Deconinck, R. Cuyvers, R. Lauwereins, J.A. Peperstraete, “Network Fault-Tolerance with Interval Routing Devices”, International Journal of Mini and Microcomputers, accepted for publication.

[21] Vounckx, J., G. Deconinck, R. Lauwereins, J.A. Peperstraete, “Deadlock-Free Fault-Tolerant Wormhole Routing in Mesh based Massively Parallel Networks”, IEEE Technical Committee on Computer Architecture (TCAA) Newsletter, Summer-Fall 1994, pp. 49-54

[22] Vounckx, J., G. Deconinck, R. Cuyvers, R. Lauwereins, “Minimal Deadlock-Free Compact Routing inWormhole Switching based Injured Meshes”, Proceedings of the 2nd Reconfigurable Architectures Workshop, Santa Barbara, California, USA, April, 1995

[23] Vounckx, J., G. Deconinck, R. Lauwereins," A Compact Fault-Tolerant, Deadlock-Free, Routing Algorithm for n-Dimensional Wormhole Switching Based Meshes", 2nd Colloquium on Structural Information & Communication Complexity, Olympia, Greece, June 12-June 14, 1995

[24] Vounckx, J., G. Deconinck, R. Cuyvers, R. Lauwereins, “Minimal Deadlock-Free Compact Routing in Wormhole Switching based Injured Meshes”, internal report KULeuven - ESAT, august 1994

[25] Vounckx, J., 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,“ *Proceedings of the HPCN Conference, Lecture Notes in Computer Science 797*, Springer Verlag, pp. 401-406, Munich, Germany, 18-20th April 1994.

[26] Wenkebach, G.: Final Report: Operator-Site Software, Report R4.3.8/A, Technical Report of ESPRIT-project 6731 (FTMPS), August 1995

[27] Wenkebach, G.: Report about System Status Visualization, Report R4.3.7/A, Technical Report of ESPRIT-project 6731 (FTMPS), August 1995