PROC. OF THE SECOND INT. CONF. ON MASSIVELY PARALLEL COMPUTING SYSTEMS (MPCS'96), IEEE Computer Society Press, Ischia, Italy, May 6-9, 1996, pp. 214-221, ISBN 0-8186-7600-0.

A Scalable Implementation of Fault Tolerance for Massively Parallel Systems

Geert Deconinck[†], Johan Vounckx[†], Rudy Lauwereins[†], Jörn Altmann[§], Frank Balbach[§], Mario Dal Cin[§], João Gabriel Silva[‡], Henrique Madeira[‡], Bernd Bieker[*], Erik Maehle[*]

[†]K.U.Leuven, ESAT-ACCA Laboratory, Kard. Mercierlaan 94, B-3001 Leuven, Belgium
Tel: +32-16-32 11 26, Fax: +32-16-32 19 86, Email: Geert.Deconinck@esat.kuleuven.ac.be
[§]F.A. Universität Erlangen-Nürnberg, IMMD III, Martensstraße 3, D-91058 Erlangen, Germany
[‡]Universidade de Coimbra, Dep. Eng. Informatica, Pinhal de Marrocos, P-3030 Coimbra, Portugal
[*]Med. Uni. zu Lübeck, Inst. Techn. Informatik, Ratzeburger Allee 160, D-23538 Lübeck, Germany

Abstract

For massively parallel systems, the probability of a system failure due to a random hardware fault becomes statistically very significant because of the huge number of components. Besides, fault injection experiments show that multiple failures go undetected, leading to incorrect results. Hence, massively parallel systems require abilities to tolerate these faults that will occur. The FTMPS project presents a scalable implementation to integrate the different steps to fault tolerance into existing HPC systems. On the initial parallel system only 40% of (randomly injected) faults do not cause the application to crash or produce wrong results. In the resulting FTMPS prototype more than 80% of these faults are correctly detected and recovered. Resulting overhead for the application is only between 10 and 20%. Evaluation of the different, co-operating fault tolerance modules shows the flexibility and the scalability of the approach.

1. Introduction

The huge number of components in a massively parallel system significantly increases the probability of a single component failure. However, the failure of a single entity may not cause the whole system to become useless. Hence, massively parallel systems require fault tolerance; i.e. they require the ability to cope with these faults that, statistically, will occur. ESPRIT project 6731 (FTMPS) implemented a *practical approach to Fault Tolerant Massively Parallel Systems* [1, 2]. In this paper, the structure of the developed FTMPS software modules and tools, their scalable implementation and their important results are explained. Section 1 explains the structure of the FTMPS modules and the target system. Besides, the fault injection experiments and field data highlight the motivations. Section 2 elaborates the different fault tolerance modules: local error detection and system level diagnosis trigger the system reconfiguration modules. The application recovery is based on checkpointing and rollback. Support for the operator is given via a set of front-end tools. For the different modules, emphasis is on the scalability of the approach and on the results. Section 3 proves how the integrated, yet modular and flexible FTMPS approach significantly improved the fault tolerance capabilities of massively parallel system: the resulting prototype is able to handle a significantly larger percentage (randomly injected) faults correctly than the initial system.

1.1 The FTMPS approach

The integrated FTMPS software modules consist of several building blocks for achieving fault tolerance as shown in Figure 1. The cooperating software modules run on the host and on the different nodes of the massively parallel target system. *Error detection* and local *diagnosis* are done on every processing element within the parallel multiprocessor. These modules run concurrently to the applications. *Application recovery* is based on checkpointing and rollback. The application itself starts the user-driven checkpointing (UDCP) or the hybrid checkpointing (HCP). These local diagnosis and checkpointing modules have counterparts running at the host: a global diagnosis module and checkpoint-controller

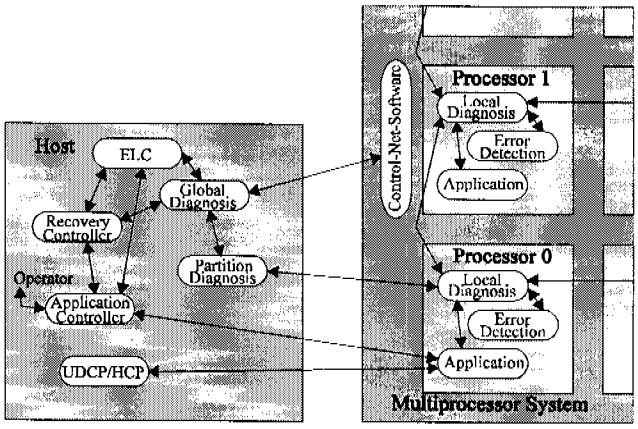

Figure 1: FTMPS building blocks.

responsible for the recovery-line management. In addition, a *recovery controller* is responsible for the system reconfiguration after a permanent failure of a component: possibly the application processes are remapped to spare nodes and new routing tables must be set up. An *interface to the operator* — the application controller (AC) — is provided by the operator site software (OSS). This OSS keeps track of the relations of failures and applications by means of the error log controller (ELC). In addition a statistical tool for the evaluation of the databases is available as well as a system visualisation tool. These different modules of the FTMPS software will be described in more detail in section 2.

The entire FTMPS software was set up to be adaptable to a wide range of massively parallel systems. Therefore, a unifying system model (USM) was introduced [1, 2]: systems that can be represented by the USM can be used as a target for the FTMPS software. The USM is based on two parts: the data-net (D-net) and the control-net (C-net). The latter one is used by the system software (initialisation, monitoring, etc.) whereas the former one is used for the applications. The D-net is divided into partitions for the applications (space sharing). Every partition consists of one or more reconfiguration entities (REs), which are the smallest entities that are used for reconfiguration. An RE can contain spare processing elements for replacing a failed node within that RE. If no (more) spares are available, the entire RE is indicated as being failed and will be replaced by an entire spare RE.

The FTMPS concepts are valid for different massively parallel systems. The prototypes of the FTMPS modules have been developed on two different Parsytec machines, the GCel-Xplorer based on a 2D-grid of T805-transputers, and the GC/PP-PowerXplorer based on a 2D-grid of PowerPC-601 and T805-transputers. These massively parallel systems are connected via a host to the user/operator environment and the disks.

In this paper, we only consider the fault tolerance aspects of the multiprocessor, and consider the host and disks to be reliable. Two considerations drive this decision. First, the number of processors (and the probability of a fault) is much larger on the massively parallel system than on the host. Second, there exist a lot of well known fault tolerance methods for uniprocessors, and to implement stable storage. Alternatively, if no fault-tolerant host is available, extra fault tolerance measures should be applied to the control-net.

The communication concept used within the target system is synchronous message passing; the processing elements are able to handle processes at least at two priority levels. Target applications come from scientific number-crunching domains without real time constraints.

1.2 Fault injection experiment as motivation for fault tolerance

In the FTMPS project, fault injection has been used to experimentally evaluate the target system. Faults were injected in the parallel machines used (Parsytec PowerPC based PowerXplorers) at the beginning and at the end of the project, so that the improvement brought by the FTMPS software modules and tools could be measured.

To inject faults a software-based fault injector was developed, called Xception. It relies on the advanced debugging facilities, included in the trap handling subsystem of the PowerPC-601 processor, and works in two phases. First, it uses the breakpoint mechanism to interrupt the normal program flow when a user-chosen trigger condition is reached (for instance, a certain address is accessed or a time-out has expired). Second, it interferes with the execution of one of the next instructions such that it simulates a fault in one of the functional units of the processor or main memory. For instance, to inject a fault in the integer arithmetic and logic unit (ALU) of the processor, Xception works as follows. When the trigger condition is reached, it executes the program in single step mode until an instruction that uses the ALU is executed (e.g. an addition), and changes the destination register in a user-specified way. A typical change is a random bit flip. Then, the program continues at full speed.

This technique has several advantages. Being totally software based, it can be easily adapted to many systems, as long as the processor used has the required built-in debug capabilities, as all modern processors do. Besides, the program subjected to the injection is executed at full speed, and does not have to be changed in any way. For a detailed description of the injector see [3].

Of the experiments made at the beginning of the project, Table 1 shows the results for two programs: Matmult, a simple program that multiplies matrices, and Ising, a bigger program that simulates the spin of particles. The outcome of the experiments was classified according to the following four categories:

- Nothing detected, *correct* results. It corresponds to those faults that are absorbed by the natural redundancy of the machine or the application.
- Nothing detected, *wrong* results. The worst situation: since nothing unusual happens, the user thinks that it was a good run, but unfortunately the output is wrong. If the results do not appear "strange" to the user, the program is not rerun.
- Error *detected*. The program is aborted with an error notification, e.g. indicating a memory protection fault.
- System *crash*. The system hangs and has to be rebooted.

	Correct	Wrong	Detected	Crash
Matmult	23%	25%	48%	4%
Ising	57%	6%	35%	2%

Table 1: Experiments with a standard machine: 3000 faults for Matmult, 4000 for Ising. All faults were transient, and consisted of two simultaneous bit flips affecting one machine instruction.

These results just show that faults indeed have bad consequences, but says nothing about the fault rate to expect in a machine. For that, we can look at statistics for the MTBF (mean time between failures) published by several computing centres that run massively parallel machines. For instance, the Oak Ridge National Laboratory (ORNL), in the USA, has published the following data about two Intel Machines, an XP/S5 with 64 processors and a XP/S150 with 1024 nodes:

	Feb. 1995	March 1995	April 1995
XP/S 5	27	11	70
XP/S 150	14	32	46

Table 2: MTBF (in hours) of some machines at ORNL (source: http://www.ccs.ornl.gov/).

On one hand, hardware failure rates cannot be directly derived from these numbers, as they also represent system crashes that are due to software faults. On the other hand, as can be seen from Table 1, crashes represent only a small percentage of fault outcomes. Besides, many software faults are "Heisenbugs" — in such complex machines, the re-execution of a program after a software induced fault is usually successful, due to slight timing changes. This means that it can be safely stated that most of the crashes reported in Table 2, and many more faults that did not lead to crashes, can be recovered by the checkpointing and restart approach followed in the FTMPS project.

More importantly, this reasoning strongly suggests that the fault rates to be expected are much higher than the crash rates reported in Table 2, meaning that the FTMPS project did indeed address a very serious problem, as the need for fault tolerance in massively parallel systems is substantial.

2. The fault tolerance modules

In this section, the different FTMPS modules and their relations are discussed. Emphasis is on the scalable approach and the significant results.

2.1 Error detection

The FTMPS project addressed hardware faults, both permanent and transient. Still, many software faults that only occur under special timing conditions are also tolerated, as long as they are detected by the existing error detection mechanisms. Indeed, when the affected programs are restarted from the last recovery-line, those timing conditions will not, in general, happen again.

2.1.1 Implementation

Error detection has been implemented in the FTMPS project under two constraints: no changes to the hardware of the used machines were allowed and the overall performance degradation could not exceed 10 to 15%. If only permanent faults would have been taken into account, then some periodic off-line tests might have done the job and could easily satisfy those restrictions; unfortunately, transient faults are much more frequent than permanent ones. Furthermore, errors caused by transient faults can only be detected by *concurrent* error detection techniques. Hence, 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. At run-time 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.

Besides the code used for the detection and correction of memory errors, six distinct categories of error detection methods (EDMs) were used in the latest FTMPS prototype:

- Built-in EDMs: Processor execution model violations (floating point exceptions, illegal instructions, illegal privileged instruction use, I/O segment error, alignment exception) and operating system level assertions. These mechanisms do not represent any overhead for the applications.
- Memory access behaviour: Detection of deviations of the proper memory access behaviour (either for instruction fetch or data load and store). This is directly implemented by the memory management unit and does not represent any overhead for the applications.
- Control-Flow monitoring: 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; to each block an ID (signature), that does not depend

on the block instructions, is assigned. Whenever a block is entered, that ID is stored in a fixed place; when a block is left, a verification is made that its ID is still stored there. Since this method requires that code to perform that storage and verification is added to the application code, it 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, one of them will be executed, thus detecting an error.

- Application level EDMs: Application-level assertions and watchdog timers. The former consist of invariants the application can verify independently of the processed data. The latter monitor 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. These methods depend on the programmer's willingness to implement them.
- Node level watchdog timer: An "I'm alive" mechanism is implemented as a part of the system-level diagnosis layer (see section 2.2), and consists of processes that periodically send messages to the processor's neighbours, to verify that all of them are still alive.
- Communication level error detection: The integrity of the messages is verified through a CRC (cyclic redundancy check).

2.1.2 Scalability and results

Since all EDMs are local to each node, these mechanisms are totally scalable.

Although 100% detection coverage is not attained, the integrated EDMs come quite close to it. Only with hardware and operating system designed from scratch we could have significantly better results. Still, the FTMPS results are quite an improvement to the initial situation. In table 3, this improvement is shown for the case of a Multigrid Solver, a large parallel program used to solve systems of linear equations.

	Correct	Wrong	Detected	Crash
without EDMs	41%	7%	37%	15%
with EDMs	28%	5%	67%	0%

Table 3: Experiments with a standard machine with and without the additional EDMs, running the multigrid application (3000 faults injected). All faults were transient, and consisted of bit flips affecting two random bits in the same 32 bit word of one functional unit, for the duration of one machine instruction, at a random time.

To better understand the results it is important to notice that the initial standard machine already made a reasonably good use of memory protection, something that does not always happen in massively parallel systems. If that were not the case, the results without the additional EDMs would have been much worse, since memory protection is a very effective behaviour based EDM.

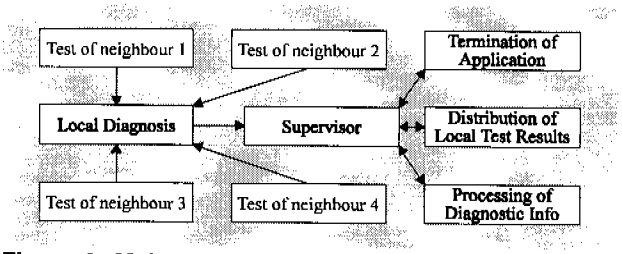

Figure 2: Main modules of the implemented system-level diagnosis algorithm.

2.2 System level fault diagnosis

The FTMPS diagnosis algorithm detects and classifies errors on system level. The first part of it is running on the **host**, and implements the highest level of the hierarchical diagnosis structure. The second level consists of the distributed diagnosis of the **data-net** and the **control-net** of the massively parallel system. On the lowest level there are modules for testing (self-testing and testing of neighboured processors). The hierarchical structure and its distributed approach make the diagnosis scalable and applicable for massively parallel systems [4].

2.2.1 Implementation

On the **host**, several diagnosis processes are running. The *global diagnosis* is started when the system is booted. It exchanges information with the other FTMPS modules — the error log controller (ELC) and the recovery controller. If an application is started on a certain partition, a *partition-wide diagnosis* module is started on the host. It communicates with the global diagnosis, and with the local diagnosis modules on the massively parallel system; besides, it tests the link connection from the host to the partition where the application is executed. Additionally, the global diagnosis has a connection to the self-checking control-net software, that is running on the control-net of the massively parallel system.

The aim of the local diagnosis of the **data-net**, is to generate a correct diagnostic image in every fault-free processor of the data-net. If this distributed diagnosis is correct, the fault-free processors can logically disconnect the faulty units from the system by stopping all communication with them.

The structure of the D-net diagnosis is shown in Figure 2. If no fault event is detected, the algorithm periodically tests the neighbouring processors. Testing is accomplished by assigning independent modules to each tested unit. This close integration of the error detection mechanisms into the diagnosis enables the event-driven

approach of the diagnosis. If one of the tests (from section 2.1, or the sending of "I'm alive" messages) detects an error in a neighbouring processor, the local diagnosis and the supervisor are informed. The latter activates the modules responsible for terminating the current application, for distributing the local test results, and for processing the diagnostic information. As the algorithm executes alternatively the local test result distribution and the syndrome decoding procedures, the diagnostic image is created gradually, taking every test outcome into consideration.

The data-net, system-level diagnosis algorithm is distributed, which makes it applicable in scalable systems; it is event-driven, e.g., only changes of the processor state will be reported. Thus it processes diagnostic information fast and efficiently, requiring only a small amount of communication and computation [5]. Therefore, the number of diagnostic messages is independent of the number of processors in the system. Employing this method, the number of tolerable faults depends only on the properties of the system interconnection topology.

In order to be able to detect and report errors within the **control-network** (self-checking control-net software), an error detecting router has been developed on top of the existing router. This allows to detect communication errors by checking the generated CRC of the messages. Crashed processors of the control-network are detected by the absence of "I'm alive" messages. Control flow errors are detected via instructions that are generated by a pre-processor [6]. When an error is detected, it is reported to the global diagnosis on the host and all affected applications are stopped immediately.

Memory faults	Processor faults	EDC-Logic faults	Link faults
Stuck-at faults	Decoding of registers	Stuck-at faults	Connection
Address-logic faults	ALU	No Correction	To/From switch
Transition faults	FPU	Wrong Detection	To/From processor
Loss of data			

Table 4: Types of faults covered by the off-line diagnosis.

These on-line test mechanisms check physically neighbouring control nodes. Therefore, a small number of control nodes can be identified where the error could have occurred. This rough localisation of the faulty components facilitates the usage of sophisticated off-line hardware tests for an exact localisation and classification of the error, because only a small number of components have to be tested (independent of the size of the control-network). The types of faults that are covered by the off-line tests implemented are shown in Table 4.

2.2.2 Scalability and results

Due to the hierarchical approach, the system-level diagnosis modules easily scale with the size of the system. The implementation of the system-level was examined, highlighting the advantages and disadvantages, in [7]. The main results are that the impact of the application on the "I'm alive" message testing mechanism is negligible and that the latency of the error detection mechanism by "I'm alive" messages can be kept small due to the small overhead caused by them. The measurement results show that the testing causes only a small overhead (less than 0.5% if the "I'm alive" messages are sent each 1.0 second).

2.3 System reconfiguration

After the error detection or diagnosis modules found a problem, the recovery controller is responsible for reconfiguring the massively parallel system and restarting the applications. The reconfiguration strategy [8] must provide each (affected) application with a partition that contains enough working processors that are able to communicate with each other. First, the different modules of the reconfiguration strategy (isolation, re-partitioning, down-loading, fault tolerant routing and re-mapping) are presented. Then we discuss their scalability and present some overhead measurements.

2.3.1 Implementation

Fault isolation at partition level is obtained by a ***double blocking mechanism***. The (re)configuration algorithm provides this when the partition borders are set up. Only if the nodes at both sides of the border are faulty, a message can cross partition boundaries.

The ***re-partitioning*** algorithm provides each affected application with a new or extended partition containing enough working processors. Since we work with massively parallel computers, the complexity of this algorithm is crucial. The developed algorithm has a complexity which is polynomially proportional to the number of allocated partitions, rather than to the number of processors in the system.

A special ***loader*** for injured systems is necessary [9] to load the application after a failure. This loader is based upon an adapted flooding broadcast mechanism. 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 FTMPS extensions) can be activated.

An important aspect of the run-time kernel is its routing functions. The ***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 based on interval routing [10, 11, 12].

The application should see a (virtually) perfect system. However, this virtually perfect system is mapped on an injured one: the ***re-mapping*** algorithm assures that the application is shielded from this by assigning each logical processor to a physical one.

2.3.2 Scalability

As this reconfiguration strategy is developed for massively parallel, from the onset scalability was taken into account. The double blocking mechanism is local. Hence it is perfectly scalable. The developed partitioning algorithm has a complexity of $O(P^2)$ with P the number of allocated partitions. Since $P^2 << N$, the number of nodes in the system, this is a good result. The fault tolerant routing algorithm is designed for compactness. The total amount of routing information per node can be reduced to O(logN.(F+n)) with F the number of failures and n the number of dimensions (here 2). The factor logN is needed to uniquely address all N nodes. The time complexity maximally increases proportionally with the number of faults in the partition. The overhead of the remapping strategy can be divided into three parts. Time overhead, data overhead and the number of unused processors. The time overhead (proportional to the number of faults) only occurs when the communication is set up. The additional amount of data is also proportional to the number of faults. Minimising the number of unused processors must be traded off against the remapping quality.

2.3.3 Results

During normal fault-free operation, no overhead is introduced for the application. Since the algorithms have been designed for scalability, the time needed for recovery is minimal: $O(F^n) + O(P^n) + O(D)$, with D the diameter of the network. The overhead during the normal operation after reconfiguration is caused by the fault tolerant routing (fewer channels available, other communication pattern) and the remapping algorithm (other communication pattern). The exact impact is very application dependent. Measurements show that, for typical applications, the overhead remains below 5%.

2.4 Application recovery

Application recovery is based on consistent checkpointing and rollback [13]. This means that periodically, the state of each process of the application is saved to a checkpoint. A set of checkpoints (one per process) which represents the consistent state of the whole application is a recovery-line. Such a recovery-line (valid set of checkpoint data) is restored after a failure: hence, the application is rolled back to a fault-free state and resumes its execution from there.

2.4.1 Implementation

The checkpoint data is saved to the disks. A checkpoint-control layer manages this checkpoint data: it builds recovery-lines from it and removes obsolete files. Consistency is guaranteed, even if failures are only detected after a (pre-defined) time, or during recovery.

Three approaches have been developed.

- In the user-driven checkpointing approach, the programmer is responsible for identifying the position of the recovery-lines in the code, and for indicating which data-items contribute to the contents of the checkpoint. Library functions are available in C and FORTRAN. The checkpoint data then consists of the state of each of these data-items. With the indication of the recovery-line in the program and the correct identification of the contributing data-items, the programmer assures consistency [14, 15].
- In the hybrid checkpointing approach, the programmer is only responsible for identifying the position of the recovery-lines in the code. The checkpoint data then consists of the whole data space of the process.
- In the user-transparent checkpointing approach, the programmer has the possibility to adjust the checkpoint interval to a value appropriate for the application and the massively parallel system. Beside this, no further action is required. With the set checkpoint interval, a daemon triggers the checkpointing; the application then freezes to assure consistency. The checkpoint data consists of the whole data space of the process [16].

These three checkpointing approaches use the same layer to send checkpoint data to the disks, and to determine and retrieve the consistent recovery-line upon rollback.

2.4.2 Scalability

The scalability of the application recovery comes from two aspects. First, the hierarchical checkpoint-control layer can (automatically or manually) be configured to optimally exploit the connection to the disks (there is no on-node disk system in our target hardware): application processes send their checkpoint data over the nearest links to the nearest disks. Only small control messages are sent between hierarchically connected controllers to assure consistency. Second, minimal run-time overhead is attained by adding some extra programming effort. In the user-driven approach, only a minimal amount of checkpoint data is saved (only those items defined by the

programmer); for the hybrid approach this amount of data is larger, but the user-involvement is smaller. The user-transparent approach does not require any user-involvement, but is more hardware dependent. The programmer or system operator can further influence the overhead by specifying how often a recovery-line should be saved.

2.4.3 Results

The user-driven and hybrid approach are integrated in the FTMPS approach. From the user's point of view, the time and storage overhead is determined by the application (i.e. how large is the checkpoint data), the hardware (what is the available bandwidth to the disks) and the MTBF of the massively parallel system (which determines an optimal time interval between consecutive recovery-lines).

The following figures are representative for the user-driven checkpointing approach. An example number-crunching application from the simulation domain is executed on 32 node system, which is connected to the disks via the host at maximal available bandwidth to disk of 1 MByte per second. The checkpoint data size is slightly more than 1 MByte per process; on the 32 node system, this corresponds to 33 MByte per recovery-line. If the MTBF of the target system is one day, then the optimal checkpoint interval is about one hour; this corresponds to a time overhead less than 1%.

2.5 Operator tools

Within FTMPS, different support tools have been developed for the operator. Conceptually, this operator site software (OSS) can be divided into an on-line part and an off-line part. The on-line part consists of the application controller (AC) and the error log controller (ELC). The database tool, statistics and system visualisation are for off-line usage, i.e. independent from the programs running at the target system.

The AC allows the operator to interface with the FTMPS modules. As such, the operator is able to keep track of the databases containing the failure list and of the status of running applications in the massively parallel system. Furthermore, the operator can send requests to the recovery software, e.g. for forcing a remapping of an application that blocks other users.

The ELC is used for the automatic recording of fault reports that are sent by the diagnosis modules. The processing of this information is done with the database tool. It manages the information coming from the diagnosis and from reports by the operator. This operator interaction allows to fill in repair reports (which components are physically replaced) and maintenance actions (e.g. system shutdowns). In order to handle the information stored in the databases, several filters can be applied for listing different failure types or components. A statistical tool is used for analysing the database entries. Important values (e.g. mean-time-to-failure (MTTF), failure inter-arrival times, etc.) can be extracted. They can be shown in different ways: bar graphs, Gantt charts, etc. This allows to analyse the dependability of the massively parallel system.

Since the presentation of the actual system status is not easy for massively parallel systems, a visualisation tool has been developed. This tool provides the operator with the possibility to view the usage of the system: the partitions of the target system are displayed with further information (idle, allocated by user X since time Y, etc.). Besides, the hardware status of the system can be displayed by colouring failed components. A hierarchical approach has been chosen where the entire system is displayed in different layers; the next level can be reached by a mouse click. An example is given in Figure 3. A graphic manager allows to adapt this tool to another target system. By labelling the components, a link to the entries in the database can be established.

The OSS tools contained within the FTMPS software provides the operator of a parallel system with arbitrary size with the ability to log failures, visualise the system status in respect to applications and failures and to show statistical measures of the system. In addition to this a possibility to manually start and stop applications is provided.

3. Conclusion

The different modules described above, have been integrated in a prototype. On this resulting prototype, we executed another set of fault injection experiments (where random faults are injected at a random time in a random processor or link unit, analogously to those described in section 1.2). This allowed to measure the improvement in dependability of this massively parallel system. In the resulting FTMPS prototype more than 80% of the faults do not cause the application to crash or produce wrong results (compared to only 40% of faults on the initial system). This means that in this case, the FTMPS

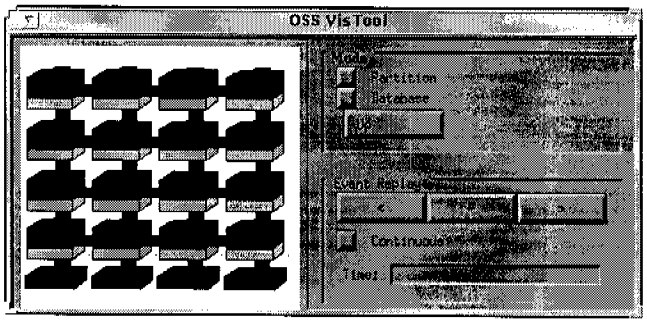

Figure 3: Visualisation tool.

modules are able to detect the errors accurately (by one of the EDMs or by the "I'm alive" mechanism after a crash), the system is properly reconfigured, and the application is restarted from the most recent, consistent recovery-line. Resulting overhead for the application is only between 10 and 20%. Although this result is far from the 100% coverage goal, it is a significant step forward from the market point of view (as shown by the field data of existing massively parallel systems). As this prototype is not yet completely stable, we are confident that fine-tuning the FTMPS modules will allow us to attain that more than 90% of the faults that are being tolerated. Higher coverages would require more extensive hardware support.

Acknowledgements

This project is partly sponsored by ESPRIT project 6731 (FTMPS): "Fault Tolerance in Massively Parallel Systems". Geert Deconinck and Johan Vounckx have a grant from the Flemish Institute for the Advancement of Scientific and Technological Research in Industry (IWT). Rudy Lauwereins is a Senior Research Associate of the Belgian Fund for Scientific Research.

4. References

[1] G. Deconinck, 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. Vichöver, "Fault Tolerance in Massively Parallel Systems", *Transputer Communications*, 2(4), Dec. 1994, pp. 241-257.

[2] J. Vounckx, G. Deconinck, R. Lauwereins, G. Vichöver, R. Wagner, H. Madeira, J.G. Silva, F. Balbach, J. Altmann, B. Bieker, H. Willeke, "The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems", Proc. of HPCN-94, Lecture Notes in Computer Science Volume 797, Springer-Verlag, Munich (D), April 1994, pp. 401-406.

[3] J. Carreira, 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 (IL), USA, Sep. 1995.

[4] Altmann, J., F. Balbach, 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 (D), Oct. 1994.

[5] Altmann, J., T. Bartha, A. Pataricza, "An Event-driven Approach to Multiprocessor Diagnosis," 8th Symposium on Microcomputer and Microprocessor Application, mP'94, pp. 109-118, Budapest (H), Oct. 1994.

[6] Hönig, J., Softwaremethoden zur Rückwärtsfehler-behebung in Hochleistungsparallelrechnern mit verteiltem Speicher, Dissertation, Univ. Erlangen-Nürnberg (D), 1994.

[7] Altmann, J., T. Bartha, 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 (D), Apr. 1995.

[8] J. Vounckx, G. Deconinck, R. Lauwereins, Reconfiguration of Massively Parallel Systems, HPCN Europe 95 conference, Milan (I), May 1995.

[9] J. Vounckx, G. Deconinck, R. Lauwereins, J.A. Peperstraete, A Loader for Injured Massively Parallel Networks, Proceedings of the 7th IASTED/ISSM International Conference on Parallel and Distributed Computing and Systems, pp. 178-180 , Washington DC, USA, Oct. 1995.

[10] J. van Leeuwen, R.B. Tan, "Interval Routing", The Computer Journal, Vol. 30(4), 1987, pp. 298-307.

[11] J. Vounckx, G. Deconinck, R. Lauwereins, Deadlock-Free Fault-Tolerant Wormhole Routing in Mesh based Massively Parallel Networks, *IEEE TCAA Newsletter*, Summer-Fall 1994, pp. 49-54.

[12] J. Vounckx, G. Deconinck, R. Lauwereins, Minimal Deadlock-Free Compact Routing in Wormhole Switching based Injured Meshes, *Proc. 2nd Reconfigurable Architectures Workshop*, CA, USA, Apr. 1995.

[13] Y. Tamir, C.H. Séquin, "Error Recovery in Multicomputers Using Global Checkpoints", *13th Int. Congress Parallel Processing*, Bellaire (MI), Aug. 1984, pp. 32-41.

[14] G. Deconinck, J. Vounckx, R. Lauwereins, "The Consistent File-Status in a User-Triggered Checkpointing Approach", *Proceedings ParCo'95*, Gent (B), Sep. 1995.

[15] G. Deconinck, J. Vounckx, R. Lauwereins, J.A. Peperstraete "A User-triggered Checkpointing Library for Computation-intensive Applications", *Proceedings Seventh Int. Conf. On Parallel and Distributed Computing and Systems*, Washington, DC, Oct. 1995, pp. 321-324.

[16] B. Bieker, G. Deconinck, E. Maehle, J. Vounckx, "Reconfiguration and Checkpointing in Massively Parallel Systems", Proc. of EDCC-1, Lecture Notes in Computer Science Volume 852, Springer-Verlag, Berlin (D), Oct. 1994, pp. 353-370.

A Scalable Implementation of Fault Tolerance for Massively Parallel Systems

Geert Deconinck[†], Johan Vounckx[†], Rudy Lauwereins[†], Jörn Altmann[§], Frank Balbach[§], Mario Dal Cin[§], João Gabriel Silva[‡], Henrique Madeira[‡], Bernd Bieker[*], Erik Maehle[*]

[†]K.U.Leuven, ESAT-ACCA Laboratory, Kard. Mercierlaan 94, B-3001 Leuven, Belgium
Tel: +32-16-32 11 26, Fax: +32-16-32 19 86, Email: Geert.Deconinck@esat.kuleuven.ac.be
[§]F.A. Universität Erlangen-Nürnberg, IMMD III, Martensstraße 3, D-91058 Erlangen, Germany
[‡]Universidade de Coimbra, Dep. Eng. Informatica, Pinhal de Marrocos, P-3030 Coimbra, Portugal
[*]Med. Uni. zu Lübeck, Inst. Techn. Informatik, Ratzeburger Allee 160, D-23538 Lübeck, Germany

Abstract

For massively parallel systems, the probability of a system failure due to a random hardware fault becomes statistically very significant because of the huge number of components. Besides, fault injection experiments show that multiple failures go undetected, leading to incorrect results. Hence, massively parallel systems require abilities to tolerate these faults that will occur. The FTMPS project presents a scalable implementation to integrate the different steps to fault tolerance into existing HPC systems. On the initial parallel system only 40% of (randomly injected) faults do not cause the application to crash or produce wrong results. In the resulting FTMPS prototype more than 80% of these faults are correctly detected and recovered. Resulting overhead for the application is only between 10 and 20%. Evaluation of the different, co-operating fault tolerance modules shows the flexibility and the scalability of the approach.

1. Introduction

The huge number of components in a massively parallel system significantly increases the probability of a single component failure. However, the failure of a single entity may not cause the whole system to become useless. Hence, massively parallel systems require fault tolerance; i.e. they require the ability to cope with these faults that, statistically, will occur. ESPRIT project 6731 (FTMPS) implemented a *practical approach to Fault Tolerant Massively Parallel Systems* [1, 2]. In this paper, the structure of the developed FTMPS software modules and tools, their scalable implementation and their important results are explained. Section 1 explains the structure of the FTMPS modules and the target system. Besides, the fault injection experiments and field data highlight the motivations. Section 2 elaborates the different fault tolerance modules: local error detection and system level diagnosis trigger the system reconfiguration modules. The application recovery is based on checkpointing and rollback. Support for the operator is given via a set of front-end tools. For the different modules, emphasis is on the scalability of the approach and on the results. Section 3 proves how the integrated, yet modular and flexible FTMPS approach significantly improved the fault tolerance capabilities of massively parallel system: the resulting prototype is able to handle a significantly larger percentage (randomly injected) faults correctly than the initial system.

1.1 The FTMPS approach

The integrated FTMPS software modules consist of several building blocks for achieving fault tolerance as shown in Figure 1. The cooperating software modules run on the host and on the different nodes of the massively parallel target system. *Error detection* and local *diagnosis* are done on every processing element within the parallel multiprocessor. These modules run concurrently to the applications. *Application recovery* is based on checkpointing and rollback. The application itself starts the user-driven checkpointing (UDCP) or the hybrid checkpointing (HCP). These local diagnosis and checkpointing modules have counterparts running at the

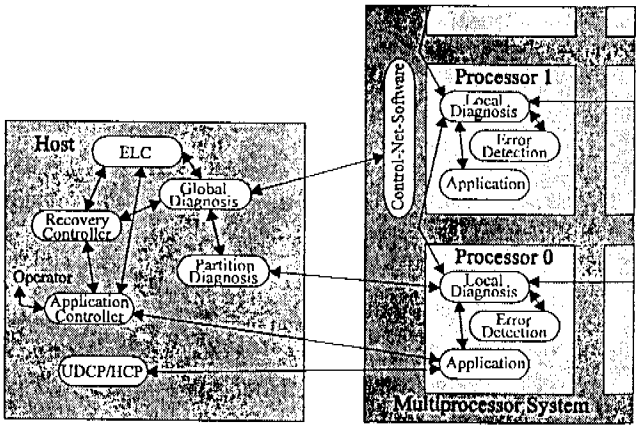

Figure 1: FTMPS building blocks.

0-8186-7600-0/96 $5.00 © 1996 IEEE

host: a global diagnosis module and checkpoint-controller responsible for the recovery-line management. In addition, a *recovery controller* is responsible for the system reconfiguration after a permanent failure of a component: possibly the application processes are remapped to spare nodes and new routing tables must be set up. An *interface to the operator* — the application controller (AC) — is provided by the operator site software (OSS). This OSS keeps track of the relations of failures and applications by means of the error log controller (ELC). In addition a statistical tool for the evaluation of the databases is available as well as a system visualisation tool. These different modules of the FTMPS software will be described in more detail in section 2.

The entire FTMPS software was set up to be adaptable to a wide range of massively parallel systems. Therefore, a unifying system model (USM) was introduced [1, 2]: systems that can be represented by the USM can be used as a target for the FTMPS software. The USM is based on two parts: the data-net (D-net) and the control-net (C-net). The latter one is used by the system software (initialisation, monitoring, etc.) whereas the former one is used for the applications. The D-net is divided into partitions for the applications (space sharing). Every partition consists of one or more reconfiguration entities (REs), which are the smallest entities that are used for reconfiguration. An RE can contain spare processing elements for replacing a failed node within that RE. If no (more) spares are available, the entire RE is indicated as being failed and will be replaced by an entire spare RE.

The FTMPS concepts are valid for different massively parallel systems. The prototypes of the FTMPS modules have been developed on two different Parsytec machines, the GCel-Xplorer based on a 2D-grid of T805-transputers, and the GC/PP-PowerXplorer based on a 2D-grid of PowerPC-601 and T805-transputers. These massively parallel systems are connected via a host to the user/operator environment and the disks.

In this paper, we only consider the fault tolerance aspects of the multiprocessor, and consider the host and disks to be reliable. Two considerations drive this decision. First, the number of processors (and the probability of a fault) is much larger on the massively parallel system than on the host. Second, there exist a lot of well known fault tolerance methods for uniprocessors, and to implement stable storage. Alternatively, if no fault-tolerant host is available, extra fault tolerance measures should be applied to the control-net.

The communication concept used within the target system is synchronous message passing; the processing elements are able to handle processes at least at two priority levels. Target applications come from scientific number-crunching domains without real time constraints.

1.2 Fault injection experiment as motivation for fault tolerance

In the FTMPS project, fault injection has been used to experimentally evaluate the target system. Faults were injected in the parallel machines used (Parsytec PowerPC based PowerXplorers) at the beginning and at the end of the project, so that the improvement brought by the FTMPS software modules and tools could be measured.

To inject faults a software-based fault injector was developed, called Xception. It relies on the advanced debugging facilities, included in the trap handling subsystem of the PowerPC-601 processor, and works in two phases. First, it uses the breakpoint mechanism to interrupt the normal program flow when a user-chosen trigger condition is reached (for instance, a certain address is accessed or a time-out has expired). Second, it interferes with the execution of one of the next instructions such that it simulates a fault in one of the functional units of the processor or main memory. For instance, to inject a fault in the integer arithmetic and logic unit (ALU) of the processor, Xception works as follows. When the trigger condition is reached, it executes the program in single step mode until an instruction that uses the ALU is executed (e.g. an addition), and changes the destination register in a user-specified way. A typical change is a random bit flip. Then, the program continues at full speed.

This technique has several advantages. Being totally software based, it can be easily adapted to many systems, as long as the processor used has the required built-in debug capabilities, as all modern processors do. Besides, the program subjected to the injection is executed at full speed, and does not have to be changed in any way. For a detailed description of the injector see [3].

Of the experiments made at the beginning of the project, Table 1 shows the results for two programs: Matmult, a simple program that multiplies matrices, and Ising, a bigger program that simulates the spin of particles. The outcome of the experiments was classified according to the following four categories:

- Nothing detected, *correct* results. It corresponds to those faults that are absorbed by the natural redundancy of the machine or the application.
- Nothing detected, *wrong* results. The worst situation: since nothing unusual happens, the user thinks that it was a good run, but unfortunately the output is wrong. If the results do not appear "strange" to the user, the program is not rerun.
- Error *detected*. The program is aborted with an error notification, e.g. indicating a memory protection fault.
- System *crash*. The system hangs and has to be rebooted.

	Correct	Wrong	Detected	Crash
Matmult	23%	25%	48%	4%
Ising	57%	6%	35%	2%

Table 1: Experiments with a standard machine: 3000 faults for Matmult, 4000 for Ising. All faults were transient, and consisted of two simultaneous bit flips affecting one machine instruction.

These results just show that faults indeed have bad consequences, but says nothing about the fault rate to expect in a machine. For that, we can look at statistics for the MTBF (mean time between failures) published by several computing centres that run massively parallel machines. For instance, the Oak Ridge National Laboratory (ORNL), in the USA, has published the following data about two Intel Machines, an XP/S5 with 64 processors and a XP/S150 with 1024 nodes:

	Feb. 1995	March 1995	April 1995
XP/S 5	27	11	70
XP/S 150	14	32	46

Table 2: MTBF (in hours) of some machines at ORNL (source: http://www.ccs.ornl.gov/).

On one hand, hardware failure rates cannot be directly derived from these numbers, as they also represent system crashes that are due to software faults. On the other hand, as can be seen from Table 1, crashes represent only a small percentage of fault outcomes. Besides, many software faults are "Heisenbugs" — in such complex machines, the re-execution of a program after a software induced fault is usually successful, due to slight timing changes. This means that it can be safely stated that most of the crashes reported in Table 2, and many more faults that did not lead to crashes, can be recovered by the checkpointing and restart approach followed in the FTMPS project.

More importantly, this reasoning strongly suggests that the fault rates to be expected are much higher than the crash rates reported in Table 2, meaning that the FTMPS project did indeed address a very serious problem, as the need for fault tolerance in massively parallel systems is substantial.

2. The fault tolerance modules

In this section, the different FTMPS modules and their relations are discussed. Emphasis is on the scalable approach and the significant results.

2.1 Error detection

The FTMPS project addressed hardware faults, both permanent and transient. Still, many software faults that only occur under special timing conditions are also tolerated, as long as they are detected by the existing error detection mechanisms. Indeed, when the affected programs are restarted from the last recovery-line, those timing conditions will not, in general, happen again.

2.1.1 Implementation

Error detection has been implemented in the FTMPS project under two constraints: no changes to the hardware of the used machines were allowed and the overall performance degradation could not exceed 10 to 15%. If only permanent faults would have been taken into account, then some periodic off-line tests might have done the job and could easily satisfy those restrictions; unfortunately, transient faults are much more frequent than permanent ones. Furthermore, errors caused by transient faults can only be detected by *concurrent* error detection techniques. Hence, 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. At run-time 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.

Besides the code used for the detection and correction of memory errors, six distinct categories of error detection methods (EDMs) were used in the latest FTMPS prototype:

- Built-in EDMs: Processor execution model violations (floating point exceptions, illegal instructions, illegal privileged instruction use, I/O segment error, alignment exception) and operating system level assertions. These mechanisms do not represent any overhead for the applications.
- Memory access behaviour: Detection of deviations of the proper memory access behaviour (either for instruction fetch or data load and store). This is directly implemented by the memory management unit and does not represent any overhead for the applications.
- Control-Flow monitoring: 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; to each block an ID (signature), that does not depend

on the block instructions, is assigned. Whenever a block is entered, that ID is stored in a fixed place; when a block is left, a verification is made that its ID is still stored there. Since this method requires that code to perform that storage and verification is added to the application code, it 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, one of them will be executed, thus detecting an error.

- Application level EDMs: Application-level assertions and watchdog timers. The former consist of invariants the application can verify independently of the processed data. The latter monitor 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. These methods depend on the programmer's willingness to implement them.
- Node level watchdog timer: An "I'm alive" mechanism is implemented as a part of the system-level diagnosis layer (see section 2.2), and consists of processes that periodically send messages to the processor's neighbours, to verify that all of them are still alive.
- Communication level error detection: The integrity of the messages is verified through a CRC (cyclic redundancy check).

2.1.2 Scalability and results

Since all EDMs are local to each node, these mechanisms are totally scalable.

Although 100% detection coverage is not attained, the integrated EDMs come quite close to it. Only with hardware and operating system designed from scratch we could have significantly better results. Still, the FTMPS results are quite an improvement to the initial situation. In Table 3 this improvement is shown for the case of a Multigrid Solver, a large parallel program used to solve systems of linear equations.

	Correct	Wrong	Detected	Crash
without EDMs	41%	7%	37%	15%
with EDMs	28%	5%	67%	0%

Table 3: Experiments with a standard machine with and without the additional EDMs, running the multigrid application (3000 faults injected). All faults were transient, and consisted of bit flips affecting two random bits in the same 32 bit word of one functional unit, for the duration of one machine instruction, at a random time.

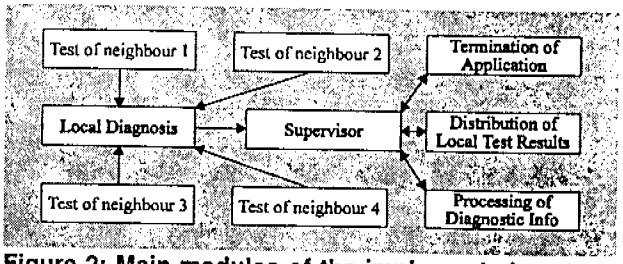

Figure 2: Main modules of the implemented system-level diagnosis algorithm.

To better understand the results it is important to notice that the initial standard machine already made a reasonably good use of memory protection, something that does not always happen in massively parallel systems. If that were not the case, the results without the additional EDMs would have been much worse, since memory protection is a very effective behaviour based EDM.

2.2 System level fault diagnosis

The FTMPS diagnosis algorithm detects and classifies errors on system level. The first part of it is running on the **host**, and implements the highest level of the hierarchical diagnosis structure. The second level consists of the distributed diagnosis of the **data-net** and the **control-net** of the massively parallel system. On the lowest level there are modules for testing (self-testing and testing of neighboured processors). The hierarchical structure and its distributed approach make the diagnosis scalable and applicable for massively parallel systems [4].

2.2.1 Implementation

On the **host**, several diagnosis processes are running. The *global diagnosis* is started when the system is booted. It exchanges information with the other FTMPS modules — the error log controller (ELC) and the recovery controller. If an application is started on a certain partition, a *partition-wide diagnosis* module is started on the host. It communicates with the global diagnosis, and with the local diagnosis modules on the massively parallel system; besides, it tests the link connection from the host to the partition where the application is executed. Additionally, the global diagnosis has a connection to the self-checking control-net software, that is running on the control-net of the massively parallel system.

The aim of the local diagnosis of the **data-net**, is to generate a correct diagnostic image in every fault-free processor of the data-net. If this distributed diagnosis is correct, the fault-free processors can logically disconnect the faulty units from the system by stopping all communication with them.

The structure of the D-net diagnosis is shown in Figure 2. If no fault event is detected, the algorithm periodically tests the neighbouring processors. Testing is

accomplished by assigning independent modules to each tested unit. This close integration of the error detection mechanisms into the diagnosis enables the event-driven approach of the diagnosis. If one of the tests (from section 2.1, or the sending of "I'm alive" messages) detects an error in a neighbouring processor, the local diagnosis and the supervisor are informed. The latter activates the modules responsible for terminating the current application, for distributing the local test results, and for processing the diagnostic information. As the algorithm executes alternatively the local test result distribution and the syndrome decoding procedures, the diagnostic image is created gradually, taking every test outcome into consideration.

The data-net, system-level diagnosis algorithm is distributed, which makes it applicable in scalable systems; it is event-driven, e.g., only changes of the processor state will be reported. Thus it processes diagnostic information fast and efficiently, requiring only a small amount of communication and computation [5]. Therefore, the number of diagnostic messages is independent of the number of processors in the system. Employing this method, the number of tolerable faults depends only on the properties of the system interconnection topology.

In order to be able to detect and report errors within the **control-network** (self-checking control-net software), an error detecting router has been developed on top of the existing router. This allows to detect communication errors by checking the generated CRC of the messages. Crashed processors of the control-network are detected by the absence of "I'm alive" messages. Control flow errors are detected via instructions that are generated by a pre-processor [6]. When an error is detected, it is reported to the global diagnosis on the host and all affected applications are stopped immediately.

Memory faults	Processor faults	EDC-Logic faults	Link faults
Stuck-at faults	Decoding of registers	Stuck-at faults	Connection
Address-logic faults	ALU	No Correction	To/From switch
Transition faults	FPU	Wrong Detection	To/From processor
Loss of data			

Table 4: Types of faults covered by the off-line diagnosis.

These on-line test mechanisms check physically neighbouring control nodes. Therefore, a small number of control nodes can be identified where the error could have occurred. This rough localisation of the faulty components facilitates the usage of sophisticated off-line hardware tests for an exact localisation and classification of the error, because only a small number of components have to be tested (independent of the size of the control-network). The types of faults that are covered by the off-line tests implemented are shown in Table 4.

2.2.2 Scalability and results

Due to the hierarchical approach, the system-level diagnosis modules easily scale with the size of the system. The implementation of the system-level was examined, highlighting the advantages and disadvantages, in [7]. The main results are that the impact of the application on the "I'm alive" message testing mechanism is negligible and that the latency of the error detection mechanism by "I'm alive" messages can be kept small due to the small overhead caused by them. The measurement results show that the testing causes only a small overhead (less than 0.5% if the "I'm alive" messages are sent each 1.0 second).

2.3 System reconfiguration

After the error detection or diagnosis modules found a problem, the recovery controller is responsible for reconfiguring the massively parallel system and restarting the applications. The reconfiguration strategy [8] must provide each (affected) application with a partition that contains enough working processors that are able to communicate with each other. First, the different modules of the reconfiguration strategy (isolation, re-partitioning, down-loading, fault tolerant routing and re-mapping) are presented. Then we discuss their scalability and present some overhead measurements.

2.3.1 Implementation

Fault isolation at partition level is obtained by a ***double blocking mechanism***. The (re)configuration algorithm provides this when the partition borders are set up. Only if the nodes at both sides of the border are faulty, a message can cross partition boundaries.

The ***re-partitioning*** algorithm provides each affected application with a new or extended partition containing enough working processors. Since we work with massively parallel computers, the complexity of this algorithm is crucial. The developed algorithm has a complexity which is polynomially proportional to the number of allocated partitions, rather than to the number of processors in the system.

A special ***loader*** for injured systems is necessary [9] to load the application after a failure. This loader is based upon an adapted flooding broadcast mechanism. 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 FTMPS extensions) can be activated.

An important aspect of the run-time kernel is its routing functions. The ***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 based on interval routing [10, 11, 12].

The application should see a (virtually) perfect system. However, this virtually perfect system is mapped on an injured one: the ***re-mapping*** algorithm assures that the application is shielded from this by assigning each logical processor to a physical one.

2.3.2 Scalability

As this reconfiguration strategy is developed for massively parallel, from the onset scalability was taken into account. The double blocking mechanism is local. Hence it is perfectly scalable. The developed partitioning algorithm has a complexity of $O(P^2)$ with P the number of allocated partitions. Since $P^2 << N$, the number of nodes in the system, this is a good result. The fault tolerant routing algorithm is designed for compactness. The total amount of routing information per node can be reduced to O(logN.(F+n)) with F the number of failures and n the number of dimensions (here 2). The factor logN is needed to uniquely address all N nodes. The time complexity maximally increases proportionally with the number of faults in the partition. The overhead of the remapping strategy can be divided into three parts. Time overhead, data overhead and the number of unused processors. The time overhead (proportional to the number of faults) only occurs when the communication is set up. The additional amount of data is also proportional to the number of faults. Minimising the number of unused processors must be traded off against the remapping quality.

2.3.3 Results

During normal fault-free operation, no overhead is introduced for the application. Since the algorithms have been designed for scalability, the time needed for recovery is minimal: $O(F^n) + O(P^n) + O(D)$, with D the diameter of the network. The overhead during the normal operation after reconfiguration is caused by the fault tolerant routing (fewer channels available, other communication pattern) and the remapping algorithm (other communication pattern). The exact impact is very application dependent. Measurements show that, for typical applications, the overhead remains below 5%.

2.4 Application recovery

Application recovery is based on consistent checkpointing and rollback [13]. This means that periodically, the state of each process of the application is saved to a checkpoint. A set of checkpoints (one per process) which represents the consistent state of the whole application is a recovery-line. Such a recovery-line (valid set of checkpoint data) is restored after a failure: hence, the application is rolled back to a fault-free state and resumes its execution from there.

2.4.1 Implementation

The checkpoint data is saved to the disks. A checkpoint-control layer manages this checkpoint data: it builds recovery-lines from it and removes obsolete files. Consistency is guaranteed, even if failures are only detected after a (pre-defined) time, or during recovery.

Three approaches have been developed.

- In the user-driven checkpointing approach, the programmer is responsible for identifying the position of the recovery-lines in the code, and for indicating which data-items contribute to the contents of the checkpoint. Library functions are available in C and FORTRAN. The checkpoint data then consists of the state of each of these data-items. With the indication of the recovery-line in the program and the correct identification of the contributing data-items, the programmer assures consistency [14, 15].
- In the hybrid checkpointing approach, the programmer is only responsible for identifying the position of the recovery-lines in the code. The checkpoint data then consists of the whole data space of the process.
- In the user-transparent checkpointing approach, the programmer has the possibility to adjust the checkpoint interval to a value appropriate for the application and the massively parallel system. Beside this, no further action is required. With the set checkpoint interval, a daemon triggers the checkpointing; the application then freezes to assure consistency. The checkpoint data consists of the whole data space of the process [16].

These three checkpointing approaches use the same layer to send checkpoint data to the disks, and to determine and retrieve the consistent recovery-line upon rollback.

2.4.2 Scalability

The scalability of the application recovery comes from two aspects. First, the hierarchical checkpoint-control layer can (automatically or manually) be configured to optimally exploit the connection to the disks (there is no on-node disk system in our target hardware): application processes send their checkpoint data over the nearest links

to the nearest disks. Only small control messages are sent between hierarchically connected controllers to assure consistency. Second, minimal run-time overhead is attained by adding some extra programming effort. In the user-driven approach, only a minimal amount of checkpoint data is saved (only those items defined by the programmer); for the hybrid approach this amount of data is larger, but the user-involvement is smaller. The user-transparent approach does not require any user-involvement, but is more hardware dependent. The programmer or system operator can further influence the overhead by specifying how often a recovery-line should be saved.

2.4.3 Results

The user-driven and hybrid approach are integrated in the FTMPS approach. From the user's point of view, the time and storage overhead is determined by the application (i.e. how large is the checkpoint data), the hardware (what is the available bandwidth to the disks) and the MTBF of the massively parallel system (which determines an optimal time interval between consecutive recovery-lines).

The following figures are representative for the user-driven checkpointing approach. An example number-crunching application from the simulation domain is executed on 32 node system, which is connected to the disks via the host at maximal available bandwidth to disk of 1 MByte per second. The checkpoint data size is slightly more than 1 MByte per process; on the 32 node system, this corresponds to 33 MByte per recovery-line. If the MTBF of the target system is one day, then the optimal checkpoint interval is about one hour; this corresponds to a time overhead less than 1%.

2.5 Operator tools

Within FTMPS, different support tools have been developed for the operator. Conceptually, this operator site software (OSS) can be divided into an on-line part and an off-line part. The on-line part consists of the application controller (AC) and the error log controller (ELC). The database tool, statistics and system visualisation are for off-line usage, i.e. independent from the programs running at the target system.

The AC allows the operator to interface with the FTMPS modules. As such, the operator is able to keep track of the databases containing the failure list and of the status of running applications in the massively parallel system. Furthermore, the operator can send requests to the recovery software, e.g. for forcing a remapping of an application that blocks other users.

The ELC is used for the automatic recording of fault reports that are sent by the diagnosis modules. The processing of this information is done with the database tool. It manages the information coming from the diagnosis and from reports by the operator. This operator interaction allows to fill in repair reports (which components are physically replaced) and maintenance actions (e.g. system shutdowns). In order to handle the information stored in the databases, several filters can be applied for listing different failure types or components. A statistical tool is used for analysing the database entries. Important values (e.g. mean-time-to-failure (MTTF), failure inter-arrival times, etc.) can be extracted. They can be shown in different ways: bar graphs, Gantt charts, etc. This allows to analyse the dependability of the massively parallel system.

Since the presentation of the actual system status is not easy for massively parallel systems, a visualisation tool has been developed. This tool provides the operator with the possibility to view the usage of the system: the partitions of the target system are displayed with further information (idle, allocated by user X since time Y, etc.). Besides, the hardware status of the system can be displayed by colouring failed components. A hierarchical approach has been chosen where the entire system is displayed in different layers; the next level can be reached by a mouse click. An example is given in Figure 3. A graphic manager allows to adapt this tool to another target system. By labelling the components, a link to the entries in the database can be established.

The OSS tools contained within the FTMPS software provides the operator of a parallel system with arbitrary size with the ability to log failures, visualise the system status in respect to applications and failures and to show statistical measures of the system. In addition to this a possibility to manually start and stop applications is provided.

3. Conclusion

The different modules described above, have been integrated in a prototype. On this resulting prototype, we executed another set of fault injection experiments (where random faults are injected at a random time in a random processor or link unit, analogously to those described in

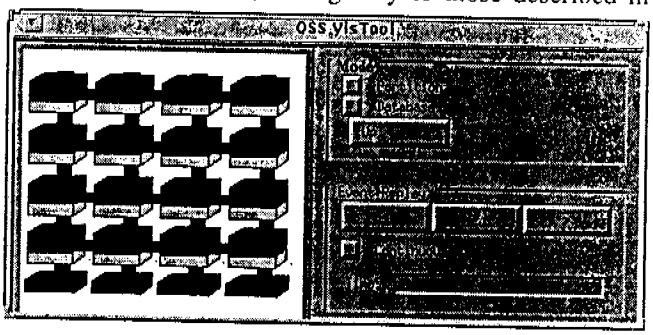

Figure 3: Visualisation tool.

section 1.2). This allowed to measure the improvement in dependability of this massively parallel system. In the resulting FTMPS prototype more than 80% of the faults do not cause the application to crash or produce wrong results (compared to only 40% of faults on the initial system). This means that in this case, the FTMPS modules are able to detect the errors accurately (by one of the EDMs or by the "I'm alive" mechanism after a crash), the system is properly reconfigured, and the application is restarted from the most recent, consistent recovery-line. Resulting overhead for the application is only between 10 and 20%. Although this result is far from the 100% coverage goal, it is a significant step forward from the market point of view (as shown by the field data of existing massively parallel systems). As this prototype is not yet completely stable, we are confident that fine-tuning the FTMPS modules will allow us to attain that more than 90% of the faults that are being tolerated. Higher coverages would require more extensive hardware support.

Acknowledgements

This project is partly sponsored by ESPRIT project 6731 (FTMPS): "Fault Tolerance in Massively Parallel Systems". Geert Deconinck and Johan Vounckx have a grant from the Flemish Institute for the Advancement of Scientific and Technological Research in Industry (IWT). Rudy Lauwereins is a Senior Research Associate of the Belgian Fund for Scientific Research.

4. References

[1] G. Deconinck, 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", *Transputer Communications*, 2(4), Dec. 1994, pp. 241-257.

[2] J. Vounckx, G. Deconinck, R. Lauwereins, G. Viehöver, R. Wagner, H. Madeira, J.G. Silva, F. Balbach, J. Altmann, B. Bieker, H. Willeke, "The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems", Proc. of HPCN-94, Lecture Notes in Computer Science Volume 797, Springer-Verlag, Munich (D), April 1994, pp. 401-406.

[3] J. Carreira, 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 (IL), USA, Sep. 1995.

[4] Altmann, J., F. Balbach, 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 (D), Oct. 1994.

[5] Altmann, J., T. Bartha, A. Pataricza, "An Event-driven Approach to Multiprocessor Diagnosis," 8th Symposium on Microcomputer and Microprocessor Application, mP'94, pp. 109-118, Budapest (H), Oct. 1994.

[6] Hönig, J., Softwaremethoden zur Rückwärtsfehler-behebung in Hochleistungsparallelrechnern mit verteiltem Speicher, Dissertation, Univ. Erlangen-Nürnberg (D), 1994.

[7] Altmann, J., T. Bartha, 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 (D), Apr. 1995.

[8] J. Vounckx, G. Deconinck, R. Lauwereins, Reconfiguration of Massively Parallel Systems, HPCN Europe 95 conference, Milan (I), May 1995.

[9] J. Vounckx, G. Deconinck, R. Lauwereins, J.A. Peperstraete, A Loader for Injured Massively Parallel Networks, Proceedings of the 7th IASTED/ISSM International Conference on Parallel and Distributed Computing and Systems, pp. 178-180 , Washington DC, USA, Oct. 1995.

[10] J. van Leeuwen, R.B. Tan, "Interval Routing", The Computer Journal, Vol. 30(4), 1987, pp. 298-307.

[11] J. Vounckx, G. Deconinck, R. Lauwereins, Deadlock-Free Fault-Tolerant Wormhole Routing in Mesh based Massively Parallel Networks, *IEEE TCAA Newsletter*, Summer-Fall 1994, pp. 49-54.

[12] J. Vounckx, G. Deconinck, R. Lauwereins, Minimal Deadlock-Free Compact Routing in Wormhole Switching based Injured Meshes, *Proc. 2nd Reconfigurable Architectures Workshop*, CA, USA, Apr. 1995.

[13] Y. Tamir, C.H. Séquin, "Error Recovery in Multicomputers Using Global Checkpoints", *13th Int. Congress Parallel Processing*, Bellaire (MI), Aug. 1984, pp. 32-41.

[14] G. Deconinck, J. Vounckx, R. Lauwereins, "The Consistent File-Status in a User-Triggered Checkpointing Approach", *Proceedings ParCo'95*, Gent (B), Sep. 1995.

[15] G. Deconinck, J. Vounckx, R. Lauwereins, J.A. Peperstraete "A User-triggered Checkpointing Library for Computation-intensive Applications", *Proceedings Seventh Int. Conf. On Parallel and Distributed Computing and Systems*, Washington, DC, Oct. 1995, pp. 321-324.

[16] B. Bieker, G. Deconinck, E. Maehle, J. Vounckx, "Reconfiguration and Checkpointing in Massively Parallel Systems", Proc. of EDCC-1, Lecture Notes in Computer Science Volume 852, Springer-Verlag, Berlin (D), Oct. 1994, pp. 353-370.

MPCS '96

Proceedings of the

Second International Conference on Massively Parallel Computing Systems

Ischia, Italy
May 6 - 9, 1996

Convened by:

Istituto di Ricerca sui Sistemi Informatici Paralleli
Istituto di Fisica Cosmica e Tecnologie Relative

In cooperation with:

EUROMICRO

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number PR07600
Library of Congress Number 96-79508
ISBN 0-8186-7600-0 (paper)
Microfiche ISBN 0-8186-7602-7

Additional copies may be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: +1-714-821-8380
Fax: +1-714-821-4641
Email: cs.books@computer.org

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM
Tel: +32-2-770-2198
Fax: +32-2-770-8505

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
JAPAN
Tel: +81-3-3408-3118
Fax: +81-3-3408-3553

Editorial production by Regina Sipple and Penny Storms
Cover by Joseph Daigle/Studio Productions
Printed in the United States of America by KNI, Inc.

The Institute of Electrical and Electronics Engineers, Inc.