An Event-driven Approach to Multiprocessor Diagnosis

Jörn Altmann[‡], Tamás Bartha[†], András Pataricza[†,‡]

[†] Department of Measurement and Instrumentation Engineering
Technical University of Budapest
Müegyetem rkp. 9
H-1521 Budapest, Hungary
email: bartha@mmt.bme.hu

[‡] Department of Computer Science III
University of Erlangen-Nürnberg
Martensstr. 3
91058 Erlangen, Germany
email: jnaltman@informatik.uni-erlangen.de

Abstract

For constructing fault tolerance mechanisms in large massively parallel multiprocessor systems, a scalable fault diagnosis is necessary, which works efficiently even if there are several thousand processors in the system. In this paper we present an event-driven, distributed system-level diagnosis algorithm, based on a general diagnosis model which does not limit the number of simultaneously existing faults. In particular, the relation between error detection and fault localization as well as two different methods for distributing diagnostic information are examined in detail. Furthermore, we give measurements concerning how does our diagnosis algorithm affect application performance.

1. Introduction

The production cost of complex, highly integrated electronic components is decreasing due to the development of manufacturing technology. As a result, **massively parallel multicomputers**, capable of operating several thousand processing elements (*PEs*) simultaneously, are gaining larger importance in computation-intensive scientific and technical applications. Yet, the huge processing capacity achieved by utilizing massively parallel architecture still requires reliable operation over a long time period. The large number of processors built in such systems increases the probability of faults, which may result in a dramatically reduced reliability without *fault tolerance*. Thus, the aim of fault tolerance is to ensure the specified operation in spite of occurring faults by preventing detected errors from becoming failures.

In design and application of massively parallel computers *scalability* is a significant requirement. A multiprocessor system is called scalable, when extending it with new resources performance increases proportionally. Due to this requirement, the application of centralized controller or observer devices is difficult, or even impossible, since they limit the number of PEs. Thus, like other parts of the system, diagnosis must be **distributed** as well. It is possible to utilize the PEs themselves for determining the system fault status: this approach is known as *distributed fault tolerance* [5][6][8][10].

The paper presents two implementations of a distributed diagnosis algorithm. The algorithm was developed for the Parsytec GCel massively parallel computer. This system uses INMOS T805 transputers as PEs, and may integrate 16’384 transputers in its full configuration. The size of the hardware can be extended in units of 16 transputers (called clusters) [11].

Scalability is achieved with a regular distributed system structure, using a two-dimensional grid interconnection network (see *Figure 1.*). Purpose of the algorithm is to generate in every fault-free transputer a correct diagnostic image, containing the fault state of system components. If the diagnosis is consistent, the fault-free transputers can logically disconnect the faulty units from the system by stopping the communication with them. Employing this method the number of tolerable faults only depends on the properties of the system interconnection topology.

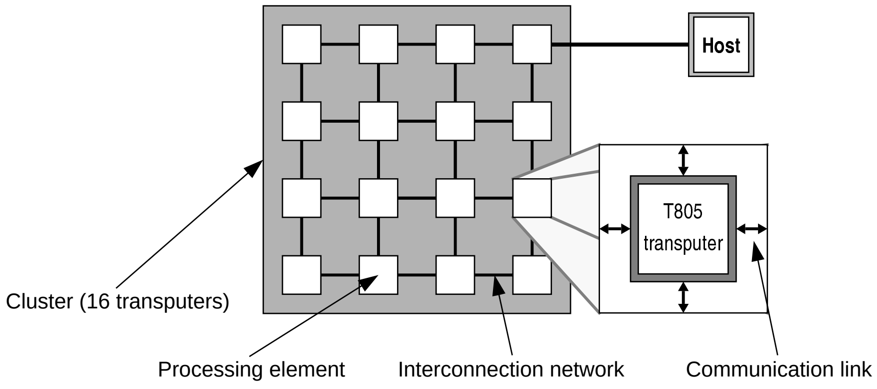

Figure 1. Structure of the Parsytec GCel

2. Diagnosis model

The application of the developed distributed system-level diagnosis algorithm (described in *Section 3.*) requires the following conditions to be fulfilled:

- **Individual and complete tests**. The algorithm assumes, that the processing elements are "intelligent units", that is they are able to perform *complete tests* (with 100% fault coverage) on units accessible via a direct communication link, independently of other PEs. Only the normal interconnection facilities may be used for testing purposes. Diagnostic messages are assumed to be protected by error-correction coding, which serves as an additional test for both the neighboring processor and the communication link connecting the PE with its neighbor. Consequently, an error in the communication facilities will also result in a bad test outcome, which enables the diagnosis of the interconnection network as well.
- **Symmetric test invalidation**. The algorithm uses the symmetric test invalidation model (*PMC*) introduced by Preparata et al. [12]. In this model, a fault-free tester always determines the condition of the device under test (*DUT*) correctly, while a test performed by a faulty tester may result in an arbitrary outcome. Since such test results do not correspond to the actual fault state of the DUT, they must be left out of consideration. Note, that the PMC model is the most general, but also the most pessimistic test invalidation model. Incorporating the PMC model, the algorithm is applicable in systems of other fault models as well. In those systems the algorithm produces correct diagnosis, but provides less diagnostic information than it is possible to obtain.
- **Diagnosability**. Majority of the diagnosis algorithms introduce an upper limit on the number of simultaneously existing faults to deal with the uncertainty caused by pessimistic test invalidation models. The underlying assumption of this *t-limit* is that few faults are more likely to occur in a properly designed multiprocessor system. The t-limit is the most number of arbitrary located faults for which the diagnosis is possible (e.g., for the two-dimensional, non-torus grid it is 2). Note, that the t-limit is a *worst-*

case diagnostic measure, in most situations it provides a much too pessimistic assumption [7].

In distributed systems with regular structure, messages exchanged between non-neighboring nodes are transferred via a chain of processors and links. Faulty processors or communication links cannot be included in this chain, because faults block the information flow. Therefore, an arbitrary set of faulty processors and links may *isolate* a group of fault-free processors if other fault-free processors in the system are unable to exchange information with this group. In such cases a diagnosis including the state of every processor in the system *cannot be generated*.

Consider a graph, in which nodes correspond to the PEs of the system, and where a directed arc exists between nodes if communication is possible between the corresponding processors in the given direction (in case of bidirectional links the arcs are not directed). This graph is called the **interconnection graph** of the system. Until this graph remains **strongly connected** (or connected in the undirected case) a diagnosis of the whole system is possible. If the graph gets unconnected by faulty processors or links, several **connected subgraphs** are created. Further on, diagnosis is restricted to the group of processors located in the same connected subgraph (the case of 3 connected subgraphs is presented in *Figure 2.*).

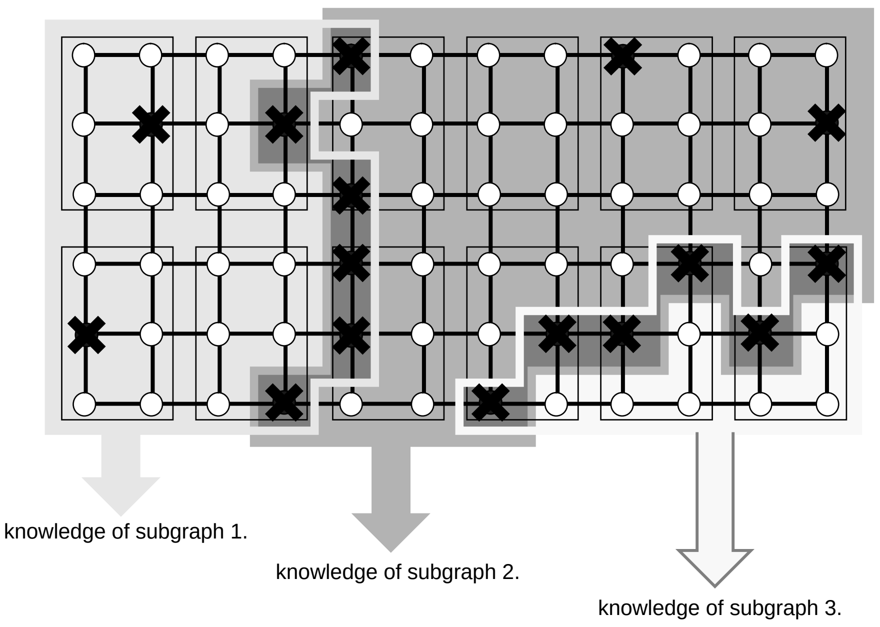

Figure 2. Diagnostic knowledge in different connected subgraphs

Note, that not the number of faulty nodes, but the size of connected subgraphs is related to the diagnostic limits. Therefore our algorithm does not require limiting the number of faults, rather it includes in the diagnosis only nodes in the same connected subgraph, classifying the state of other processors as *unknown*. Each fault-free PE makes* diagnosis about its own subgraph, as indicated in *Figure 2.*.

- **Determining the real message order**. The arrival of messages at a processor will not always correspond to the order of their creation, due to communication delays. Such a situation can occur if a PE becomes faulty during the testing process. Then, messages received in an incorrect order will cause the algorithm to generate an incorrect diagnosis. To avoid this, logical time-stamps related to test execution must be attached to the diagnostic messages, and the real order of messages must be determined using a distributed event-ordering procedure [9].

3. Diagnosis algorithm

During distributed diagnosis, processors test the *neighboring* (i.e., accessible via a direct edge in the system interconnection graph) processors. Every PE generates a local diagnostic image, which it transfers to the tested fault-free neighboring nodes. Communication stops, when all of the fault-free processors transferred their local test results to all other fault-free units. All processors have obtained information about the fault state of each other node (unless some faulty PEs cut the system into different isolated components, see *Section 2.*). This information is reliable, since it was forwarded over a chain of fault-free nodes. Processing the incoming local diagnostic images all PEs can determine the fault state of the whole system.

Note, that the diagnostic process is almost identical for the different processors (only the inhomogeneity at the two-dimensional grid borders has to be taken into consideration), so each processor uses an identical diagnosis algorithm. The algorithm consists of two phases: an *initial* and a *working* phase. Two observations motivated to split the algorithm:

1. High current during switching on/off may damage the electronic components of the system. Hence, the majority of faults occurs (or already exists) in the initial period. During further operation the failure rate is expected to be lower.
2. In the initial period processors do not have information about the condition of other components (i.e., other PEs and communication links), so the whole diagnosis process must be completed once. Later the system fault state does not change significantly compared to the starting diagnostic image. Therefore significant communication and administration can be saved by calculating and distributing only the *differences* between the current (diagnosed) fault state and the stored diagnostic image.

3.1 Initial phase of the algorithm

After generating the local diagnostic image, the interprocessor communication starts, and it continues until each fault-free processor has received local diagnostic images from all the others. Every PE sends its information to its neighbors, but further on it only receives and forwards the messages sent by other units. To evaluate the termination rule of the communication process, PEs must keep a record of the incoming messages to determine from which nodes they have not received diagnostic image yet. For this purpose they must also discover which units are accessible via a path of fault-free processors and links.

Using our synchronous communication protocol, there are processors that meet the termination condition before others, due to inherent inhomogeneity of the two-dimensional grid topology (i.e., some neighbors of processors located on grid edges are missing) and obstacles in communication formed by faulty components [4]. These processors must inform their neighbors before termination, otherwise the neighbors would possibly try to communicate with the terminated processor, thus causing a deadlock situation. To avoid it, he algorithm has a *termination* period. In this period ready-to-terminate PEs send special messages to their neighbors, so the still active nodes will not communicate with these PEs further on. When the information was sent to each neighbor, processors decode the received syndromes using the algorithm described in *Section 4.*, thus completing the initial phase.

The initial phase of the diagnosis algorithm is integrated into the booting and loading process of the transputer system.

3.2 Working phase of the diagnosis algorithm

After finishing the initial phase, the algorithm continues with the working phase. At this point all processors have an initial, system-level diagnostic image. During further operation processors periodically test their neighbors, and compare the obtained test results to the stored diagnostic image. If they find a difference (indicating a new fault occurrence or an on-line repair), they update the local diagnostic image and start broadcasting messages, containing the test results. As the test result distribution and syndrome decoding process is initiated by the changes in the local diagnostic image, the working phase of the diagnosis algorithm is **event-driven**.

4. Fault localization

In both phases of the algorithm it is necessary to analyze the obtained diagnostic information, and to decide the fault state of each accessible unit of the system. This task is accomplished by a syndrome decoding algorithm (defined by the state diagram in *Figure 3.*), which describes the different states of determining the classification (*faulty*, *fault-free*, or *link fault*) of a unit under diagnosis. In each state the momentary classification is shown between parentheses, state transitions indicated by arrows are influenced by the received test results.

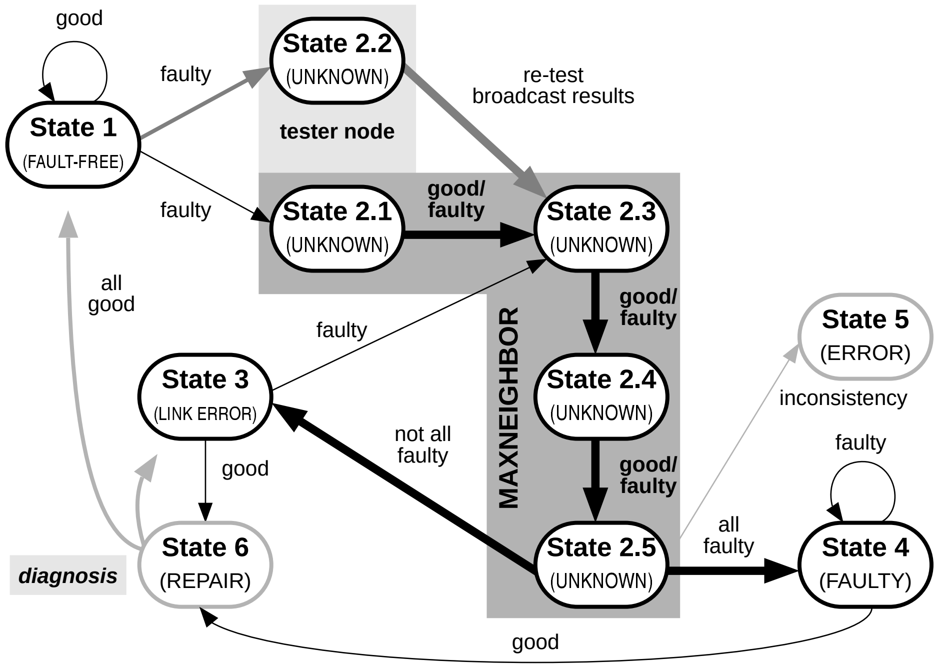

Figure 3. State diagram of the syndrome decoding algorithm

State 1 is the initial state: here the classification remains *fault-free*, until a message indicating a fault in the unit has been received from one of its testers. Depending on whether the PE performing the diagnosis is tester of the examined unit or not, the algorithm executes State 2.2 or State 2.1. During the subsequent four states (marked by gray background) test results from the other testers of the unit are analyzed. The number of these states (indicated by MAXNEIGHBOR in *Figure 3.*) equals to the number of tester units.

In these states the classification of the unit under diagnosis is *unknown*. Decision is made when all messages were evaluated (State 2.5). If every tester of the examined unit found it to be faulty, then the unit itself is assumed to be faulty (State 4). If only some of the testers detected a bad test outcome, then the corresponding communication links between these testers and the tested unit are assumed to be faulty (*link fault*, State 3). State 6 provides the possibility of taking on-line repairs into consideration. Here an extra diagnostic process is required to assure the consistency between the diagnostic images stored at processors located in different connected subgraphs (recall *Section 2.*), if links or nodes previously isolating these processors were repaired. State 5 indicates if there is inconsistency between the corresponding test results of

PEs testing each other (both PEs were diagnosed as fault-free, yet one of them states that the other is faulty). This state is included as a verification option only. If the assumptions introduced in *Section 2.* are true it will never be entered.

5. Two implemented approaches for the working phase

The implementation of the working phase can be done in different ways, depending on how the processing power of the PE is divided between the diagnostic process and the running application. In the following we describe two alternative implementation approaches. The first one (called *"Separate testing phase"* approach) uses "rough" tests requiring only a fraction of the processing capacity, thus yielding more computational power to the application. The second (called *"Gradual syndrome decoding"* approach) uses more precise and complete tests; yet it results in a decreased application performance due to the more intensive diagnosis process. The other main difference between the approaches is in the termination rule used in the distribution of local test results: the first approach is based on a time-out rule, which is safer and easier to evaluate, but makes the distribution phase longer than waiting until the necessary local test results were received, which rule is incorporated in the second approach.

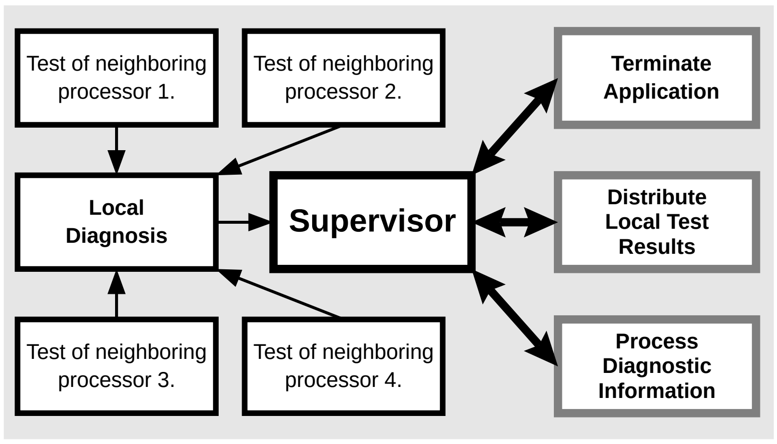
Test of neighboring processor 1.
Test of neighboring processor 2.
Terminate Application
Local Diagnosis
Supervisor
Distribute Local Test Results
Test of neighboring processor 3.
Test of neighboring processor 4.
Process Diagnostic Information

Figure 4. Main modules in the implemented algorithm

The main structure of both implementation approaches for the working phase of the algorithm are shown in *Figure 4.* If no fault event is detected, the algorithm periodically tests the neighboring processors. Testing is accomplished by assigning independent threads to each tested unit [2].

5.1 The "Separate testing phase" approach

If the tests detect an error at 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 calls the procedures responsible for terminating the current application, for distribution the local test results, and for processing the diagnostic information (as described in *Section 4.*) [1].

– **Test of neighboring processors**. In this approach, each testing module is comprised of three threads: one for receiving <I'm alive> messages from the corresponding neighboring processors, one for sending such messages, and one for measuring the delay between two consecutive <I'm alive> messages received from the same neighbor. If this delay exceeds a given limit, the thread sends an error message to the module performing local diagnosis.

– **Terminating the application**. The task of this module of the diagnosis algorithm is to stop the execution of the application on all the PEs as soon as possible. This is necessary to prevent the propagation of errors. If the application is quickly stopped, the error propagation probability is reduced, because no further communication – except the communication of the diagnostic information – will take place.

  For quick termination of the application, the module initiates a fast broadcast. The broadcast messages are received on every node by the *local diagnosis* module, which starts the exception handling. For the implementation of a fast broadcast no specific routing mechanism is required; the existing routing mechanism of the Parsytec GCel system supplemented with a high-level, fault-tolerant communication protocol is fully sufficient.

– **Distribution of local test results**. After terminating the application, the module for distributing the local diagnosis results is activated. At first, neighbors of the faulty processor start a separate testing phase, executing different tests in order to locate the cause of the error. The possible causes are faulty links and faulty processor components. These tests even make possible to classify faults as *temporary* or *permanent*. The outcome of the tests constitutes the local diagnosis results.

  The module transfers the local diagnosis results by a fault-tolerant broadcast to the supervisor module of all processors. This fault-tolerant broadcast is similar to the one introduced above under *"Terminating the application"*.

5.2 "Gradual syndrome decoding" approach

In case of an occurring error (detected by failure of a local test or a bad test outcome received from another PE), the appropriate testing thread or the *communication router* module notifies the *supervisor* module. The supervisor module first terminates the running application to avoid error propagation, then it starts the module which distributes the local test results. During communication every tester of the faulty node broadcasts its test results, and the fault-free units analyze these results [3].

– **Test of neighboring processors**. This approach is based on *direct* testing. Direct testing may take two forms: either the tester processor sends some stimulus to the unit under test, which sends back its responses to the stimulus; or the testing facilities are built-in the tested PE, and only a request-reply message exchange between the tester and the built-in testing unit is required to obtain the current fault state.

  The testing module is composed of two threads: one for sending data or requests to the tested units, and one for receiving the replies. The sending thread maintains a queue about the pending requests and the corresponding request start times. The receiving thread measures the delay between every request start and reply arrival, deletes the entries related to incoming messages from the queue, and detects timed-out messages.

– **Distribution of local test results**. When one of the testers of a faulty node detects the error, it sends the bad test outcome to all of its fault-free neighbors. The neighbors forward the message to their neighbors (excluding the sender) and so on, until the message arrives at every accessible fault-free node. To minimize the number of redundant messages during the broadcast, a routing mechanism is used, which prevents the nodes from receiving the same message in multiple copies.

  Receiving an error indication about a node, processors first check whether the report corresponds to one of their neighbors (precisely, to a node which they are assigned to test). If they are testers of the node, they re-test it and after forwarding the original report they broadcast the test result as well. This way every fault-free processor receives up-to-date test results from all testers of the examined node.

On obtaining a test result, the distribution module calls the procedure responsible for processing the diagnostic information. This way distribution of local test results and processing of diagnostic information are executed alternatively. The diagnostic image is produced gradually, utilizing the same tests for error detection and localization.

5.3 Comparison of the two approaches

The main differences between the two described implementation approaches, their advantages and disadvantages are the following:

- **Testing mechanism**. As mentioned above, the “Separate testing phase” approach uses <I’m alive> messages to detect fault occurrences, and has a separate testing phase. While <I’m alive> messages are easy to process, these tests are less dependable, and more post-processing is required later. As the application is quickly stopped on detecting a fault, there is enough time to perform more detailed tests in the following testing phase. The direct tests used in the “Gradual syndrome decoding” approach provide better fault coverage, but reduce the application performance in a larger amount than <I’m alive> messages.
- **Fault classification**. An advantage of the separate testing phase of the first approach is the possibility to classify faults as permanent or transient. Also some specific tests may be performed in the dedicated testing phase (e.g., for testing particularly the interconnection link, CPU, memory, etc. of the DUT), it is possible to obtain more detailed information about the examined node (to localize a faulty component). Such classification is not possible in gradual diagnosis, since decisions are made on the basis of tests used also for error detection (i.e., there are no repeated tests).
- **Termination rules**. The two approaches use different criterion for terminating the distribution phase. The “Separate testing phase” approach waits for a certain time interval, in which all of the local test results must be received. This simple method makes the algorithm robust against losing messages. On the other hand, determination of the optimal time-out limit requires exhaustive experiments with the given machine. The “Gradual syndrome decoding” approach waits only until local test results from each tester of the faulty node were received, therefore it can be faster than the other one.

6. Measurements

As stated in *Section 3.*, during the operation of the system faults seldom occur. In a fault-free system the event-driven diagnosis algorithm performs only error detection, thus the testing mechanism has the largest impact on the application performance. We have examined the application run-time overhead caused by testing. The minimal run-time overhead can be achieved using the <I’m alive> message testing method, therefore the “Separate testing phase” approach was measured. In *Figure 5.* the run-time overhead is given, running one application and the diagnosis algorithm concurrently on each processor. Performance measurements with various applications, which differ in the amount of communication, were made. However, the shape of curves describing the overhead corresponding to the different applications are so identical, that only one curve was given in *Figure 5.*

Figure 5. states that the overhead is inversely proportional to the time between two consecutive <I’m alive> messages. The sending of <I’m alive> messages has a very little impact on the application run-time, if the interval between two messages is longer than one second. If the <I’m alive> messages are sent in every 500 milliseconds, the overhead is bigger, but does not exceed 0,2 percent. Even if the <I’m alive> messages are sent in every 6 milliseconds, the overhead is still less than 2 percent.

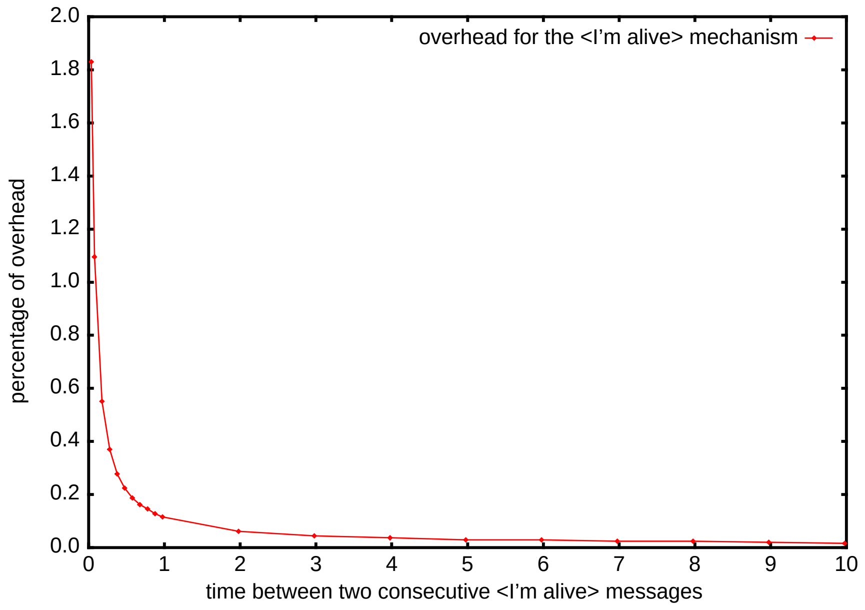

Figure 5. Application run-time overhead caused by testing, using the <I’m alive> mechanism

7. Conclusions

In this paper we introduced a new system-level diagnosis algorithm. The 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.

The general structure of the algorithm, consisting of two separate phases has been described. A new syndrome decoding method, which produces the diagnosis gradually was given. Furthermore, we presented an extended diagnosis model, which makes possible to obtain all accessible diagnostic information without limiting the number of tolerated faults within the system.

Additionally, we presented two implementations based on the algorithm. One of them uses different tests for error detection and localization, using a separate testing phase after quick termination of the running application. The other 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. The two implementations were compared, highlighting the advantages and disadvantages in both of them. We have included some measurement results, which show that the testing causes only a small overhead, assuring that the implementations will work efficiently in real systems.

Our future work will concentrate on the relation between the tests for error detection and the tests for error localization.

References

[1] Altmann, J., “Diagnoseprotokolle in Multiprozessorsystemen,” *Diploma Work*, University of Erlangen-Nürnberg, February 1993.

[2] 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*, Berlin, October 1994.

[3] Bartha, T., “Diagnostic Algorithms of Multiprocessor Systems,” *Diploma Work*, Technical University of Budapest, 1993.

[4] Behr., P. M., W. K. Giloi, and W. Schröder, “Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems,” *Aspects of Computation on Asynchronous Parallel Processors*, pp. 239-247, North-Holland, 1989.

[5] Bianchini, R., R. Buskens, and M. Stahl, “On-Line Diagnosis in General Topology Networks,” *Proc. IEEE Workshop on Fault-Tolerant Par. and Distr. Systems*, pp. 114-121, Amherst, July 1992.

[6] Dal Cin, M., and F. Florian, “Analysis of a fault-tolerant distributed diagnosis algorithm,” *IEEE Proc. 15th Int. Symp. on Fault-Tolerant Computing*, pp. 159-164, Ann Harbor, June 1985.

[7] Kime, C. R., “System Diagnosis,” in *Fault-Tolerant Computing: Theory and Techniques*, Prentice-Hall, New York, pp. 577-623, 1985.

[8] Kuhl, J. G., and S. M. Reddy, “Distributed Fault-Tolerance for Large Multiprocessor Systems,” *ACM-Sigarch Newletter 8,* no. 3, pp. 23-30, 1980.

[9] Kuhl, J. G., S. M. Reddy, and S. H. Hosseini, “On Self Fault-Diagnosis of the Distributed Systems,” *IEEE Proc. 15th Int. Symp. on Fault-Tolerant Computing*, pp. 30-35, Ann Harbor, June 1985.

[10] Meyer, F.J., and G. Masson, “An efficient fault diagnosis algorithm for symmetric multiprocessor architecture,” *IEEE Transaction on Computer*, vol. EC-27, pp. 1059-1063, November 1978.

[11] Parsytec Computer GmbH., “The Parsytec GCel Technical Summary,” Version 1.0, Aachen, 1991.

[12] Preparata, F., G. Metze, and R. Chien, “On the Connection Assignment Problem of Diagnosable Systems,” *IEEE Transaction on Computer*, vol. EC-16, no. 6, pp. 848-854, December 1967.