Tolerating Transient Faults in MARS *

H. Kopetz, H. Kantz, G. Grünsteidl, P. Puschner, J. Reisinger

Institut für Technische Informatik
Technische Universität Wien
Treitlstr. 3/182/1, A-1040 Vienna, Austria

Abstract: Transient faults form the dominant fault class in many computer systems. Because of their transient nature and short duration, it is difficult to detect and locate transient faults. This paper discusses the concepts of transient fault handling in the MARS architecture. After an overview of the MARS architecture, the mechanisms for the detection of transient faults are discussed in detail. In addition to extensive checks in the hardware and in the operating system, time redundant execution of application tasks is proposed for the detection of transient faults. The time difference between the effective and the maximum execution time of an application task is used for this purpose. Whenever a transient fault has been detected, the affected component is turned off and reintegrated immediately by retrieving the uncorrupted state of the actively redundant partner component. In order to reduce the probability of spare exhaustion (in the case of permanent faults) "shadow components" are introduced. The reliability improvement, which can be realized by these techniques, is calculated by detailed reliability models of the architecture, where the parameters are based on experimental results measured on the present MARS prototype implementation.

1 Introduction

Fault tolerant systems must provide their specified services despite the occurrence of faults in the system's components. In some critical applications, e. g. flight control or train signaling and control, a single incorrect output can trigger a catastrophe. In such an application an extremely high level of error detection coverage is required. Since the number of errors generated is proportional to the duration of a transient fault, the probability of error detection increases with the fault duration [5]. Detection of errors caused by short transient faults, i. e. faults with a duration of less than 100 machine cycles, is particularly difficult.

Even the duplication of hardware resources and the immediate comparison of the results, as realized in [9], may not suffice to detect transients which are caused by the same external event. The problem of this approach is, that two redundant hardware units operating in full synchronism (i. e. driven by the same clock), may fail in correlated failure modes if disturbed by a single external event (e. g. a local power spike). We are convinced that a certain amount of temporal displacement between redundant computations is required in order to realize a very high error detection coverage in such a situation.

In this paper we propose the time redundant (two-fold) execution of critical tasks and the comparison of their results for error detection. Transients which are shorter than the task execution times will be detected by this technique. Transients which are longer than the task execution time will result in so many subsequent errors, that the probability of detecting any one of them is very high.

To reduce the severity of failure modes, many error detection mechanisms have been implemented. In order to avoid the problem of spare exhaustion, the installation of "shadow components" is proposed.

The effects of these techniques for the handling of transient faults on the overall system reliability are analyzed by detailed reliability models.

This paper is organized as follows: After a brief presentation of the basic principles of our reference architecture in Section 2, the hardware mechanisms as well as the operating system and application software mechanisms for the detection of transient faults are discussed in Section 3. Section 4 focuses on error handling and introduces the concept of "shadow components". Markov models for analyzing the reliability of the resulting fault-tolerant architectures are described in Section 5. Finally, the results of this analysis are presented in Section 6.

2 The MARS Architecture

MARS (**MA**intainable **R**eal-Time **S**ystem) is a fault-tolerant distributed real-time architecture for hard real-time applications. One of its basic characteristics is the completely deterministic behavior even under specified peak load conditions. In order to achieve this determinism, MARS is strictly time driven and periodic [11].

Fault-tolerance in MARS is based on fail-silent components running in dual active redundancy and on sending each message twice on the two actively redundant real-time busses. A "well behaved" component produces either correct results or no results at all, i. e. as soon as it detects an error within itself, it turns itself off. This concept provides

*This work was supported in part by the ESPRIT Basic Research Project 3092 "Predictably Dependable Computer Systems"

CH 2877-9/90/0000/0466/$01.00 – 1990 IEEE

fault-isolation (prevention of error propagation). As long as at least one of the set of redundant components operates, the full service can be maintained.

A MARS system consists of a set of clusters. Each cluster is composed of several autonomous components, i. e. selfcontained computers including the application software (set of real-time tasks), which are loosely coupled by redundant synchronous real-time busses (MARS-busses). Each component consists of a single board computer and links to the real-time busses. The network is reduced to passive channels.

Each component contains a local real-time clock. All clocks of good components are synchronized with a known maximum deviation Δ (in the order of μsec) by a fault-tolerant synchronization algorithm [13]. This global time is used to realize a TDMA-strategy (time division multiple access) controlling the access to the communication channel and to verify the validity of messages.

Communication among tasks and components is realized by the exchange of broadcast "state"-messages, i. e. every message sent by any component is received by all other components. State messages are not consumed when read and a new version of a state message updates the previous version. Each message contains a validity time attribute. As soon as the current time reaches the validity time of a message, the message is discarded by the operating system.

Furthermore, each message contains a signature which is calculated at the application level. We assume that each receiving task checks the signature of an incoming message. If the result of this check is negative, the message is discarded. This additional check is introduced in order to provide an end-to-end error detection mechanism [18].

An identical copy of a network operating system (the MARS operating system) runs locally and autonomously in each MARS component and guarantees the deterministic system behavior.

3 Detection of Errors

The fault hypothesis in MARS includes transient and permanent physical faults in the components and on the real-time busses. Transient faults may be caused by alpha particles, cosmic rays, high operational and environment temperature, short deviations of the supply voltage, etc., i. e. faults in the physical universe are assumed [2]. Design-faults are not considered in this paper.

Each operational component contains two data structures, which are called i-state and h-state [10]. The i-state (initialization state) specifies the content of the read/write memory that is not changed during a computation. It consists of the programs and permanent data of a component. The i-state is static and can be protected by a checksum and memory protection (read-only or execute-only access). The h-state (history state) comprises all relevant information which has been accumulated in the history (i. e. the sequence of computations since the initialization) of the component. The h-state is dynamic and can be changed by a computation. It is evident that the h-state depends on the granularity of computation and has to be initialized when starting the component. We distinguish between faults causing errors in the i-state and in the h-state.

Concerning the communication system we assume that faults may occur in the incoming or outgoing communication link of a component or on the busses. Since a fault on a bus has the same effect as the fault on an outgoing link, the transient failure rates of the busses are subsumed in the failure rates of the outgoing links. We further assume that a message contains a sufficient amount of redundancy (check sum) in order to detect any mutilation of the message contents. To summarize, the reliability of the communication system is described by message loss rates of the incoming and outgoing link of a component, respectively.

In the following subsections we will discuss the error detection mechanisms and the fault assumptions at the hardware, the operating system and the application software level.

3.1 Hardware Error Detection

MARS components are designed with the goal of achieving a high self-checking coverage. For determining the self-checking coverage of the components, extensive experiments have been performed [6]. Based on these experiments, we assume, a coverage better than 99%, if extensive error detection (e. g. watchdog timers, power supply monitoring, temperature monitoring) and conservative hardware design are provided.

Since the hardware error detection coverage for transient faults is considerably lower than that for permanent faults [5] and the corresponding failure rate is higher, we are using techniques in MARS to increase the self-checking coverage for transient faults.

More than 3 billion messages have been sent via the real-time bus of an existing MARS cluster to experimentally evaluate the message loss rate [15]. Based on these results we concluded that the message loss rate can be subsumed in the component failure rate, provided that a membership service protocol [12] is implemented in the operating system [7].

3.2 Operating System Error Detection

Failures which are not detected by the hardware (less than 1%) have to be detected by the operating system and the application software. Since the operating system only manipulates messages, (it is not supposed to change the contents or the signature of any message), the probability that a transient interfering with the operating system will result in a mutilated message with a correct signature is very low.

Based on our experience with the prototype implementation, we assume that 1/3 of the processing time is consumed by the operating system and 2/3 is consumed by the application software.

The following error detection mechanisms are implemented in order to detect errors at the operating system level:

- Use of robust storage structures[1] to check the structural consistency of the system data structures during run time [19, 4]. Although this method is, due to its design overhead, hardly suitable for application tasks, it is appropriate for the operating system, since it will not change for different applications.
- Massive use of assertions. The parameters of each function and local variables are checked against predefined boundaries at various points in the code.
- Plausibility tests for the results of system routines.
- Check of the execution time of system routines.

We make the pessimistic assumption that 90% of the errors that are not detected by the hardware, will be detected by the operating error detection mechanisms.

In order to handle faults causing the loss of messages on the links or on the channel a membership protocol is implemented at the operating system level (see also Section 4.1). Generally, a component needs to know about the traffic on the channel and the behavior of the other components to detect this kind of failure. This knowledge is contained in the membership protocol. Without this knowledge a component which does not hear any traffic on the channel can never decide whether there is no traffic, or one of the senders' outgoing link or its own incoming link have failed.

3.3 Application Software Error Detection

Transient faults occurring during application software execution are detected by comparing the results produced by executing tasks in time redundancy, i. e. the operating system attempts to execute each task twice at different times.

Since MARS tasks are scheduled statically (i. e. processor time is assigned to each task before run time) an upper bound for the worst case execution time (MAXTc, i. e. calculated maximum execution time) of each task has to be calculated. This is done by a MAXT analysis tool which analyzes application programs and computes their worst case execution time with respect to the preconditions imposed by the MARS environment.

The MAXT analysis operates on high-level language statement level. It does not perform any semantic analysis. Consequently, it cannot extract the full knowledge of the application specific characteristics of a program. Although new constructs have been introduced allowing programmers to explicitly express more application specific knowledge in their programs [14], a task's MAXTc will in most cases be significantly higher than the real execution time for given input data. This implies that the reserved processor time will not be fully utilized in most cases.

This remaining processor time, i. e. the time between the actual execution time and the MAXTc of a task, is used for a second, time redundant execution of the task, thus allowing the detection of transient faults. In fact the length of a time slice reserved for each task is set to a value that assures 99 % certainty that the second execution of the task will complete, following the first one. This duration ($t_{99\%}$) is derived from the results of intensive task testing.

In order to guarantee the time slice for a worst case task execution and to reach the 99 % coverage for a time redundant computation, the length of the time slice reserved for a task is set to max(MAXTc,$t_{99\%}$).

If the second execution of the task completes, the signatures of the results are compared and in case of a mismatch, the component is turned off.

[1] A storage structure is called robust if some (specified) number of changes made to structural information can be detected [3].

4 Error Handling

4.1 Fail Silent Mechanism

Whenever an error is detected, the first error handling activity is to turn off the affected component. Therefore, all error detection mechanisms described so far attempt to establish the assumed self-checking behavior of a component.

In order to perform a consistent error reporting mechanism, a membership service is realized inside the MARS operating system. At any point in time $t_{observe}$ the membership service provides each component with consistent and timely knowledge about the operational state of the other components at a past point in time t_{past}, i. e. every active component knows at time $t_{observe}$ which components were active at time t_{past}. Under the considered fault hypothesis the time interval between $t_{observe}$ and t_{past} is bounded and has the length of two TDMA cycles [12]

A task residing on a component is provided with the membership information by the operating system. This membership information is used by the error handling mechanisms described below.

4.2 Shadow Component

After a component has failed, it takes some time (in the range of a few seconds in the case of an h-state failure to some days in the case of a permanent hardware failure) before it is "repaired" and can continue its normal work. Since each fail silent component runs in dual active redundancy, the redundant component still maintains the full services after a single failure. However, a further single failure can lead to a system crash during this period.
There are two evident ways to handle this situation:

- Reducing the interval of nonredundant operation by short maintenance intervals, which significantly increases the maintenance costs.
- Using triple redundancy instead of dual redundancy. Since there has to be a TDMA slot for each active component, in the case of triple redundancy, the response times of tasks are increased up to 50% of the original response times under dual redundancy.

MARS introduces the concept of shadow components, which

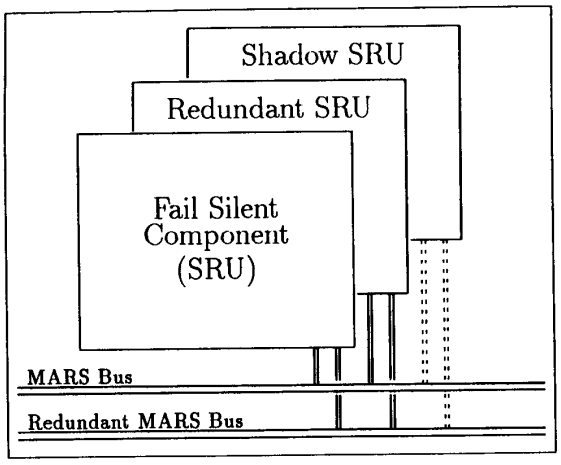

Figure 1: Fault tolerant unit

combines the advantages of the two conventional concepts.

A shadow component operates in redundancy with two other actively redundant components. It receives and processes messages, and therefore maintains an internal state equivalent to the states of the active components. The difference from the other components is that no TDMA slot is assigned to it. Consequently, it does not broadcast any message on the MARS busses. In case of a failure of an active component (detected by the membership protocol), the TDMA slot of the faulty component will be assigned to the shadow component. Since there is no need to reload the h-state, the switch to the new component can be performed immediately after a component failure has been detected by the membership protocol.

Since shadow components do not send any messages, the membership protocol cannot detect their failures. The consequences of a shadow component failure are as follows:

- If the failure is transient, the component restarts automatically. This failure has no effect on the cluster, because the active components are not affected.
- If the failure is permanent, the component cannot be restarted. A permanent component failure is repaired at the end of the current maintenance interval. There is no need to inform the other components of the cluster about this failure.

In order to allow the detection of permanent outgoing link failure of a shadow component, each shadow component should become active at least once in each maintenance cycle. In normal operation two actively redundant components and one shadow component form a fault tolerant unit (FTU), see Figure 1. As long as a single component of the FTU is operational it will deliver the specified service. The smallest replaceable unit (SRU) is given by a component.

4.3 Component Restart

After a component has shut itself down due to an error, it tries to perform a restart, starting with a check whether a permanent or a transient failure has occurred. This is done by test routines, checking RAM and ROM, the processor and the controller chips to peripheral devices.

After determining a component to be non-faulty i. e. no permanent failure has occurred, the next step is to determine whether the failure occurred in the h-state or in the i-state. Both the program and the read-only data segment are provided with checksums. By testing them, a failure in the i-state can be detected. In this case, the first action is to reload the component's i-state. Since there is no TDMA slot assigned to the restarting component, it cannot make a request to get its i-state. One special component, called the maintenance component, of the MARS cluster periodically broadcasts the i-state of each component on the busses. The failed component recovers its i-state using these messages, which may take several minutes.

After the correct i-state has been reestablished (either by reloading it or by assuming its correctness as a consequence of detecting no checksum error), the h-state is restored. Since each active component sends its h-state on the bus periodically, the component is recovered by accepting the relevant h-state of the redundant partner component. If there are already two actively redundant components, the restarted one works as a shadow component, otherwise the component becomes an active one.

5 Reliability Model

The primary objective of the reliability model is to examine the reliability of a MARS system and to check whether the reliability requirement can be met under the architectural concepts and techniques selected. Secondly, on a more detailed level, it should point out reliability bottlenecks, critical failure types and the sensitivity of system parameters.

In this model, we critically examine different probabilities of an efficient, correct handling of faults and adequate maintenance policies in order to achieve an balance between maintenance cost and reliability.

5.1 Basic Model

A MARS system consisting of one cluster, with 8 FTUs, is examined. Since MARS clusters operate almost independently, MARS clusters are modeled as independent units. More clusters may be modeled as a series/parallel diagram.

We model two types of architectures. In the first type, a FTU consists of two actively redundant components without a shadow component. In the second alternative, a shadow component is added to each FTU to increase the degree of redundancy.

The following assumptions and idealizations have been made to get a mathematically tractable model:

- Faults occurring in one component are statistically independent of faults occurring in other components.
- Three different exponentially distributed failure modes are considered for components:
  - Transient failures corrupting the i-state of a component, with failure rate λ_i.
  - Transient failures corrupting the h-state with failure rate λ_h.
  - Permanent failures with failure rate λ_p.

  Based on our experiences with the existing MARS system we assume the following values for the failure rates: Transient failures occur with rate $\lambda_t = 1/5000\ h^{-1}$; permanent failures occur with rate $\lambda_p = 1/50000\ h^{-1}$.

  Because of the size of the i-state and the h-state we choose $\lambda_i = (1/5) * \lambda_t$ and $\lambda_h = (4/5) * \lambda_t$.

  Due to the membership protocol the transient bus failure rate is subsumed in the failure rate of the component. Using two actively redundant busses, the permanent bus failure rate is omitted.
- The three different types of component failures require different repair activities.
  - A corrupted i-state requires a reboot of the i-state, which is performed periodically via a maintenance component. The time to reboot transient i-state failures is exponentially distributed with rate $\mu_i = 12\ h^{-1}$
  - If the h-state is corrupted, only a new initialization of the component, with rate $\mu_h = 120\ h^{-1}$ is necessary.
  - For the repair of permanent failures, two different approaches are examined. First, maintenance on demand is carried out with a constant repair rate ranging from $\mu_p = 1/4\ h^{-1}$ to $\mu_p = 1/24\ h^{-1}$. In the second alternative periodic maintenance is investigated with maintenance intervals from 1 to 4 weeks. In this type of model, we consider only the interval between two consecutive repair activities omitting the repair time of a permanent failure of a component. This assumption is justified because repair is done online. In contrast to maintenance on demand, where just an exchange of the faulty component is carried out, we assume that extensive system tests are carried out in the case of periodic maintenance and that the system is in a fault free state afterwards.
- The probability that a failure is covered is described through a coverage parameter c. We pessimistically assume that permanent failures have the same coverage parameter as transient failures. This self-checking coverage parameter is defined as the proportion of failures which would have occurred in a fault-intolerant system that are successfully averted by means of fault-tolerance [1]. Covered failures cause a component to be turned off. With probability $1 - c$ a failure is not covered. In this case we pessimistically assume that the system crashes.

  As already discussed in Section 3, we assume the following coverage values: The hardware has a self-checking coverage of 99%. We further assume that the operating system can handle 90% of the operating system failures not covered by the hardware, and two-fold execution of application tasks fixes 99% of the remaining application software failures. Assuming 1/3 of the failures affect the operating system and 2/3 affect the application software we achieve a coverage of 0.993 for single execution of application tasks and 0.9996 in the case of a two-fold execution of application tasks.
- An operating maintenance component is required for restoring the i-state of a failed component. Since the maintenance component operates actively redundant, we assume that transient i-state failures can always be restored with rate $\mu_i = 12\ h^{-1}$.
- All repair activities due to transient and permanent failures are assumed to be perfect.
- Since shadow components do not send any messages on the bus, we assume that shadow components failures cannot lead to FTU failures.

Thus, we model the MARS cluster hierarchically using structural decomposition, resulting in 8 subsystems each describing a FTU. The single FTUs are modeled as a Markov chain. Depending on the chosen assumptions with respect to use of shadow components and to the repair policy of permanent faults we derive 4 different Markov chains. Exemplarily, we show the Markov chain for a fault tolerant unit with a shadow component and periodic maintenance intervals in Figure 2. The interested reader can find an elaboration of the other models in [8].

States representing operational system states are termed $aa|x$ or $a|xx$ $x \in \{a,i,h,p\}$ (a = working, i = transient i-state failure, h = transient h-state failure and p = permanent failure). The number of a's to the left of the bar indicates the number of active components in the FTU. The letters to the right of the bar represent the state of the inactive components. We distinguish between two failure states *Fse* and *Fc*. *Fse* represents failures caused by spare exhaustion, while *Fc* represents lack of coverage failures, caused by an imperfect handling of faults.

It is assumed that active components and shadow components have the same reliability behavior. Since shadow components do not broadcast any results, a wrong result cannot cause a system failure. We derive a transition rate $2 * c * \lambda_y + \lambda_y$ from state $aa|a$ to state $aa|y$, $y \in \{i,h,p\}$. The first term describes the covered failure rate of the two active components, the second term describes the shadow component failure rate. With rate $2 * (1-c) * \lambda_g$, $\lambda_g = \lambda_i + \lambda_h + \lambda_p$ a failure in state $aa|a$ is not covered and causes a system

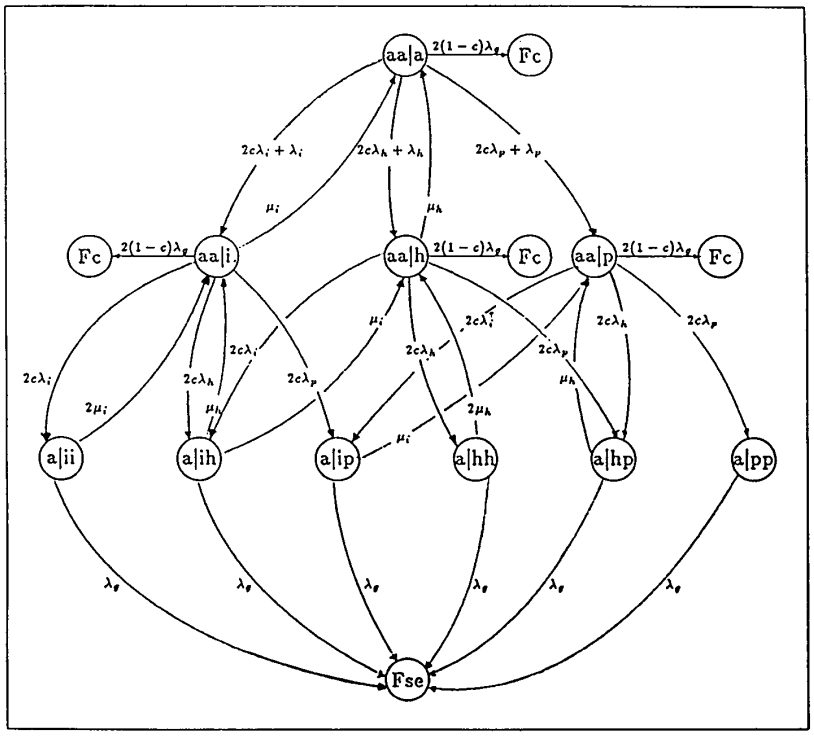

Figure 2: Markov model with shadow component

failure.

Transient i-state (h-state) component failures are recovered in parallel with rate μ_i (μ_h).

Since permanent component failures are repaired periodically, these repair activities are not incorporated in the Markov graph, in contrast to the maintenance on demand model, where an exponentially distributed repair time is assumed.

A MARS cluster is then modeled by combining the Markov graphs representing a FTU as a series/parallel block diagram.

The models have been evaluated with the dependability evaluation tool SHARPE [16, 17], which provides a hybrid, hierarchical modeling framework for Markov models as well as Combinatorial models.

6 Discussion of the Results

In this section, the 4 presented alternatives (shadow components / no shadow components and single / two-fold execution of application tasks) are compared and discussed. We concentrate on the analysis of the reliability improvement by the proposed techniques. Two-fold execution of application tasks improves the self-checking coverage. Shadow components reduce the probability of spare exhaustion failures.

Two different repair strategies for permanent component failures have been analyzed. To achieve a short mean time to repair (MTTR), maintenance on demand exchanging the faulty component is used. In the second alternative periodic maintenance is investigated. In this case we assume that after repair the system is in a fault free state.

For the following examples, we modeled a MARS cluster with 8 FTUs, where all FTUs are assumed to be necessary for proper operation.

Table 1 shows the calculated mean time to failure (MTTF) of such a MARS cluster in years, for a maintenance on demand with MTTR 4h and 24h, respectively.

It is one major objective of the MARS architecture to achieve low maintenance costs. The impact of changing the maintenance policy using periodic maintenance is shown in Table 2.

The significant difference between maintenance on demand and periodic maintenance in columns 3 of Table 1 (MTTR = 24 hours) and Table 2 (maintenance interval = 1 day) is caused by the higher probability of spare exhaustion failures in the case of maintenance on demand.

Let us have a more detailed look at the impact of two-fold execution of application tasks.

Figure 3 shows the reliability function of the MARS cluster without using shadow components as a function of the

	without shadows single execution	shadows single execution	without shadows two-fold execution	shadows two-fold execution
4h	4.53y	4.63y	57.01y	81.07y
24h	4.08y	4.63y	23.89y	81.01y

Table 1: MTTF of the MARS cluster for maintenance on demand

	without shadows single execution	shadows single execution	without shadows two-fold execution	shadows two-fold execution
1d	4.34y	4.64y	36.56y	81.05y
7d	3.18y	4.63y	8.83y	79.97y
28d	1.75y	4.62y	2.63y	66.82y

Table 2: MTTF of the MARS cluster for periodic maintenance

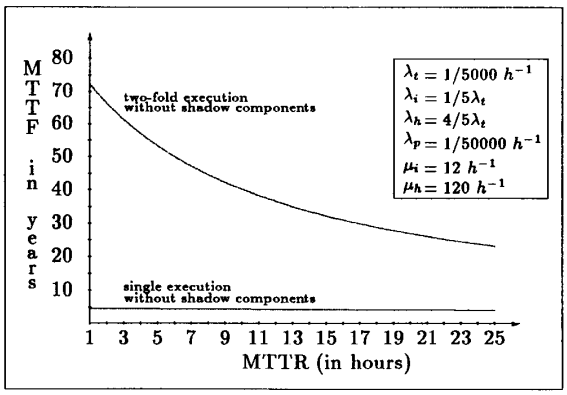

Figure 3: Single execution vs. two-fold execution

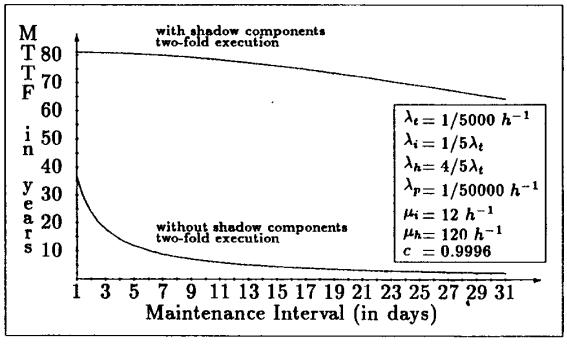

Figure 4: Without shadow component vs. shadow component

MTTR. In the system without two-fold execution of application tasks, lack of coverage failures become the dominant failure source even for a short MTTR. A change of the MTTR scarcely influences the system reliability. Under the given assumptions a significant reliability improvement can be achieved by improving the self-checking property of the components. However, in this case a short MTTR is required to achieve a balance of the two failure modes, spare exhaustion and lack of coverage. Such short repair times require high maintenance costs on the order of 30 % of the initial hardware costs per year.

In Figure 4 the reliability functions for the two alternatives shadow component and no shadow component are compared. For the alternative without shadow components, spare exhaustion is the major failure source, resulting in a low MTTF for maintenance intervals on the order of weeks. Reducing the probability of this type of failure by means of shadow components leads to a significant increase of the system's MTTF and allows maintenance intervals on the order of weeks.

Introducing shadow components increases the initial hardware costs by 50 % but reduces the maintenance costs from 30 % (for repair on demand) to about of 5 % of the initial hardware costs per year.

7 Conclusion

Two techniques for improving the reliability of distributed systems have been presented: the time redundant execution of application tasks to increase the error detection coverage and the installation of shadow components to reduce the probability of spare exhaustion. The effects of these two techniques on the reliability of a typical MARS cluster have been analyzed by a reliability model. Based on experimental data observed on the existing MARS prototype a significant reliability improvement can be expected by the implementation of these techniques. The two-fold execution of critical tasks increases the MTTF by a factor of more than 10, provided a short maintenance interval of about 4 hours is guaranteed. The introduction of shadow components makes it possible to increase the maintenance

interval from 4 hours to 30 days without adversely affecting the system reliability.

8 Acknowledgement

We gratefully acknowledge the useful comments and suggestions on earlier versions of this paper by A. Damm, P. Ezhilchelvan, G. Leber, R. Schlatterbeck, W. Schütz, A. Steininger, A. Vrchoticky, and R. Zainlinger.

References

[1] T. Anderson, P. Barrett, D. Halliwell, and M. Moulding. Software Fault Tolerance: An Evaluation. *IEEE Transactions on Software Engineering*, 11(12):1502–1510, Dec. 1985.

[2] A. Avizienis. The Four-Universe Information System Model for the Study of Fault-Tolerance. In *Proc. 12th Int. Symposium on Fault-Tolerant Computing*, pages 6–12, Santa Monica, California, June 1982.

[3] J. Black, D. Taylor, and D. Morgan. A Compendium of Robust Data Structures. In *Proc. 11th Int. Symposium on Fault-Tolerant Computing*, pages 129–131, Portland, Maine, June 1981.

[4] A. Damm. The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems. In *Proc. 16th Int. Symposium on Fault-Tolerant Computing*, pages 171–176, Vienna, Austria, July 1986.

[5] A. Damm. *Experimental Evaluation of Error-Detection- and Self-Checking Coverage of Components of a Distributed Real-Time System.* PhD thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Oct. 1988.

[6] A. Damm. Self-Checking Coverage of Components of a Distributed Real-Time System. In *Proc. 4th International Conference on Fault-tolerant Computing Systems*, pages 308–319, Baden-Baden, Germany, Sep. 1989.

[7] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The Operating System of MARS. *ACM SIGOPS Operating Systems Review*, 23(3):141–157, July 1989.

[8] H. Kantz. Dependability Aspects for the Design of the MARS System. Research Report 19/89, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1989.

[9] V. Kini. Fault Tolerance in the BIIN Computer System. In *Proc. International Workshop on Hardware Fault Tolerance in Multiprocessors*, pages 1–6, Univ. of Illinois, Urbana, USA, June 1989.

[10] H. Kopetz. The Failure Fault (FF) Model. In *Proc. 12th Int. Symposium on Fault-Tolerant Computing*, pages 14–17, Santa Monica, California, June 1982.

[11] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[12] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. In *Int. Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, CA, USA, Aug. 1989.

[13] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[14] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[15] J. Reisinger. Failure Modes and Failure Characteristics of a TDMA driven Ethernet. Research Report 8/89, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Oct. 1989.

[16] R. Sahner and K. S. Trivedi. Performance and Reliability Analysis using directed acyclic Graphs. *IEEE Transactions on Software Engineering*, 13(10):1105–1114, Oct. 1987.

[17] R. A. Sahner and K. S. Trivedi. Reliability Modeling using SHARPE. *IEEE Transactions on Reliability*, 36(2):186–193, June 1987.

[18] J. Salzer, D. Reed, and D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[19] D. J. Taylor and J. P. Black. Guidelines for Storage Structure Error Correction. In *Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pages 20–22, Ann Arbor, Michigan, June 1985.