Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 733 – 737

ISBN 978-83-60810-14-9
ISSN 1896-7094

Reliability of Malfunction Tolerance

Igor Schagaev
Dept. of Computing
London Metropolitan University
166-220 Holloway Road, London N7 8DB, United Kingdom
Email: i.schagaev@londonmet.ac.uk

***Abstract*—Generalized algorithm of fault tolerance is presented, using *time*, *structural* and *information* redundancy types. It is shown that algorithm of fault tolerance might be implemented using hardware and software. It is also shown that for the design of efficient fault tolerant system elements must be malfunction tolerant. The advantage of element malfunction tolerance is proven in reliability terms. Reliability analysis of a fault tolerant system is performed with deliberate separation of malfunction and permanent faults. Special function of reliability gain is introduced and used to model system reliability. Maximum reliability of fault tolerant system is achievable with less than a duplication system and depends on malfunction/permanent fault ratio and coverage of faults. System level of fault tolerance prerogative is the reconfiguration from permanent faults.**

I. Introduction

THE world of applications for fault tolerant systems is expanding beyond our imagination. Areas where we need them most are called embedded systems. The principal features of on-board systems as well as embedded safety critical systems are Fault Tolerance (FT) and Real Time (RT) response. They both are reflected in the main system components: hardware (HW) and system software (SSW). The process of FT system design consists of two mutually dependent sequences of HW and SSW development, using different redundancy types [1]. The design of the FT system assumes that the required specification is already known, including faults that system should tolerate [2],[14],[25].

FT system design required objectives are: performance, reliability and low power consumption (the latter especially for spaceborne computers). All of them are achievable in combination by means of a careful and balanced introduction and monitoring of hardware and system software features. Achieving reliability assumes either using of multiple unreliable components [3], or higher reliability components, or application of various types of internal redundancy to maximize efficiency of the solution [4],[5].

The term *fault tolerant system* needs a rigorous definition; at the same time it must be differentiated from such terms as *graceful degradation* and *fail-stop systems* . Avizienis [6-10], Laprie [9,11-12] and Siewiorek [14] proposed that a system be called *fault tolerant* (FTS) if it recovers itself to full performance or at least continues with sufficient specified functions and required features.

This work has been done as a part in the ONBASS (On Board Active Safety System) Project No. 160045, funded by the EC.

The *gracefully degradable system* (GDS) is the system that can recover itself and continue functioning in degraded mode after occurrence of a fault [12] . In turn, if a system can stop itself correctly once a fault has been detected in acceptable state it is called a *fail-stop system* (FSS) [13].

The most critical reliability requirements for on-board applications are MTTF (Mean Time To Failure) 10-25 years and system availability not less than 0.99 over the whole life cycle of the system, typical for aircraft, satellites, gas pipelines, etc.

II. Generalized Algorithm of Fault Tolerance

Several authors [2],[5] proposed to consider fault tolerance as a process of several steps for: proving that the appeared fault did not damage hardware, determination of type of fault (malfunction or permanent), checking the consistency of software state, detecting of correct states and further recovery from the correct state.

This process was called a Generalized Algorithm of Fault Tolerance (GAFT). The *information* , *time* and *structural* types of system redundancy enable to complete these steps of the process of fault tolerance. Combination of GAFT and possible redundancy types forms a matrix shown in Table 1. All steps of GAFT are implemented by SSW and HW. Steps {A,…,G} vs. redundancy types form a framework for the classification, design and analysis of implemented FT computer systems. According to Table 1, a system is called *fault tolerant* if it implements GAFT, i.e. every row (line) was visited and the algorithm is completed. There is no doubt that the algorithm can be completed differently, using different redundancy types at each step. This shows that fault tolerant systems may differ in:

- The time taken to implement various steps
- Types of redundancy used
- Types of faults which might be tolerated.

The taxonomy of Table 1 might be used for comparison of various design solutions of FT systems. The other interesting property of this taxonomy is an evaluation of efficiency of redundancy in implementation of various steps of GAFT. Therefore, this taxonomy might provide analytic evaluation and assist in selection of the most efficient solution for the implementation of on-board system from the alternative approaches.

Table I.
GAFT vs. redundancy types

Steps	Redundancytypes					
	hw			*sw*		
	i	*s*	*t*	*i*	*s*	*t*
A.Provetheabsenceof fault, ELSE						
B.Determinetypeof fault						
C.If fault is permanentTHEN						
D.Reconfigurehardware						
E.Proveconsistencyof softwareELSE						
F.Locatefaultystates						
G.Recoversoftware						

Description of a fault tolerant system proposed initially for on-board systems [5] is more rigorous. It applies an algorithmic definition of the feature of *fault tolerance* using GAFT. In other words, GAFT introduces FT not as a feature but as *a process*.

We consider a system as *fault tolerant if and only if* it implements GAFT transparently for an application. On-board system's main function is the implementation of control algorithms; we call it "process one" or P_1 [15]. Thus the functional definition of fault tolerance for control computers is the following: if the system provides for an application ("process one") full functionality and can transparently for the application recover itself from a predefined set of faults, this system will be *fault tolerant* .

The transparency of fault tolerance for the applications (for on board computers) means that GAFT is completed within a defined time (between sequential data outputs or inputs; in practice it is about 10-125 milliseconds). In short, the system is fault tolerant if it provides failure-free-mode for the implementation of the process P_1 . Again, it is assumed that the performed recovery is transparent for the application.

III. Time Redundancy and Fault Tolerance

The GAFT implementation varies in terms of redundancy types used and, therefore, the time to complete. In terms of time slot which is required to implement FT of the system, different levels of granularity of SW might be used: instruction, procedure, module, task and system, as presented in Fig. 1.

At the *instruction level* scheme, SSW assumes an elimination of a fault appearing and its influence within the instruction execution. For example, the triplicate memory voter masks the faults of one memory element and therefore, this fault is tolerated at the *instruction level* .

In turn, the techniques such as microinstruction or instruction repetition can be considered as implementation of instruction level FT for the processor. From the system point of view an implementation of fault tolerance at the *instruction level* has a serious and undisputable advantage: fault checking, detection and recovery can be completely transparent to the system software.

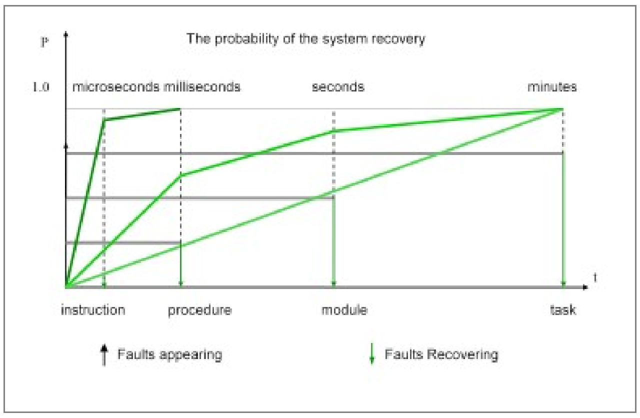

Fig. 1 Levels of implementation of fault tolerance

At the *procedure level* an implementation of fault tolerance assumes that a fault and its influence is tolerated and eliminated within the scope of a procedure execution. For example, the techniques such as *recovery blocks* (wrongly known as recovery points) [16],[17],[18],[19] can be considered as procedure- oriented scheme. This approach requires system software support (compiler level and run-time system) and might be efficient if and only structural programming approach is used (known after [20]).

The analysis of efficiency of procedure level approach together with aspects of implementation was done in [21]-[23]. There is no doubt that the state of the program must be conserved/re-generated to recover from a hardware fault. For this scheme the volume of the information required to restore the system after fault is much larger (and has significant software overheads).

For the *module level* scheme a module tolerates influence of a fault. For example, module restart or run of a simplified alternative module [10] might be used. Module recovery reduces the probability to restart the task or the whole system. The same comments apply for the procedure scheme.

At the *Task level* a fault and its influence is eliminated by restart of a task after reconfiguration of hardware.

Initialization of this scheme of GAFT implementation might also use time redundancy available within task execution. Clearly this option has the highest software, state space and time overheads for implementing fault tolerance. This level requires system software (operating system) support for HW fault tolerance. Details of the SSW features to support hardware fault tolerance for parallel FT systems are discussed in [24].

System level of fault tolerance assumes automatic or manual (when the control process allows it) restart of the system after fault and performing of control function after full restart of the system. Automatic restart has to be supported by system software using automatic hardware testing and software restart schemes.

IV. Performance of FT Systems

The time impact of implementing GAFT at each of these levels is different, as well as is the delays caused by their use. The current embedded system practice indicates the following order of magnitudes for timing:

- microseconds for the instruction level
- milliseconds for the procedure level
- hundreds of milliseconds for the module level
- tens of seconds at the task level
- ten of minutes to hours at the system level.

The various schemes of GAFT implementation have different overheads, capabilities for tolerating of various classes of faults, power consumption overheads and affiliated system costs. For the instruction level scheme the concrete hardware support is required. For example, duplicate or triplicate hardware modules could result in serious overhead in power consumption and size. Some better solution was recently found as well (http://www.itacsltd.com).

However, not all hardware sub-systems need necessarily be designed in this way. For example, a cost benefit analysis (where 'cost' infers financial, power consumption, chip area, time delays, etc.) might indicate that it is worthwhile to have the processor and system RAM fault tolerance, achieving fault tolerance of the rest of the system using procedure or module schemes.

The *procedure* and *module level* schemes require much less hardware support but, of course, have a larger timing and software-coding overhead. For the *module* and the *task level* schemes with no extra hardware, system software support for fault tolerance is actually incomplete, as any *permanent* hardware fault would cause the system stop and loss of the control application. If the system requires reboot to recover then it is resilient to *malfunctions* (transient faults), not the permanent faults.

Depending on the level employed in the implementation of fault tolerance, systems differ in time required of achieving it (see the thick lines in Fig. 1). Note that different fault types might be tolerated by different schemes applied in combination; in other words, levels of implementation of fault tolerance are not mutually exclusive.

A good fault tolerant system tolerates the vast majority of malfunctions within the *instruction* execution making them invisible in terms of other instructions (and software) . The malfunctions with longer time range might be detected and recovered differently, for example, at the procedural or task level.

The complexity of GAFT implementation also differs in the types of fault that have to be tolerated. Even knowing that the ratio of malfunctions is the order of magnitude higher than permanent faults, it is necessary to implement the special schemes for re-configurability and recoverability of the hardware to eliminate an impact of *permanent faults* on the system.

At the other extreme, the system could tolerate the vast majority of *malfunctions* at the task level using SW. The operating system support or even user support also might be used to tolerate hardware faults. From the user's viewpoint even 'WINTEL' systems (Windows/Inter based) might be considered as fault tolerant as long as scheduled application was completed and results delivered on time. In this case, the fault tolerance was achieved by means of maintenance of hardware and involvement of system engineers to fix the fault 'in time'.

Most WINTEL systems assume system reboot and restart of the applications is an acceptable way of operation. Practical experience of real-time systems confirms that this is not the case, at least for systems that must run continuously for more than a few hours.

V. Reliability Evaluation for Fault Tolerance

From the classic reliability point of view any extra redundancy of hardware reduces the absolute reliability of the system [26]. At the same time, a reliability of the system might be increased if introduced redundancy is able to tolerate faults of the main part of the system by means of GAFT and itself contributes less in the system fault ratio.

Thus, part of the problem – the decreasing reliability of the system caused by hardware redundancy at some point becomes part of solution. GAFT implementation breaks down to two processes for checking (P2) and recovery (P3), while process (P1) is application execution [15]. Clearly, there are design tradeoffs to be made to achieve the optimum operational reliability and redundancy types used.

Reliability analysis [27] introduces a reliability value for each element (hardware redundancy) and assumes a Poisson failure rate λ . This makes possible to calculate the overall reliability as a function of time for the whole system, if the structure of the system is known.

Hardware redundancy used at the various steps of GAFT degrades in reliability over time; thus achievable performance and reliability and their degradation within life cycle of the RT system are dependent. Therefore, an analysis of the surface shape and evaluation of performance and reliability degradation caused by the redundancies used should be performed for every fault tolerant system. Fig.2 presents qualitatively a slope where a fault tolerant system should be located, between the plane of requirements and curves of reliability and performance degradation.

Furthermore, the introduction of the cost to implement each proposed solution makes it possible to summarize the system overheads required to implement fault tolerance. There is no doubt, a quantitative evaluation of reliability, performance and cost overheads within one framework might be extremely efficient for justification of the design decisions and comparison of different approaches in implementation of fault tolerance.

There is a correspondence between reliability of FT systems and steps of GAFT related to the malfunction tolerance. The next sections analyze this correspondence.

A.Impact on Reliability of Malfunction Tolerance

Let's denote a rate of permanent fault for the system without redundancy as $\boldsymbol{\lambda}_{pfl}$, the malfunction rate as $\boldsymbol{k}\,\boldsymbol{\lambda}_{pfl}$. For modern technologies $\boldsymbol{k}$ varies from 10^3 to 10^5 (the latter applies to aeronautics). The probability of operation without permanent fault within time gap [0,T) and mean time to failure are determined by formulas (1).

$$P_1(t) = e^{-((1+k)\lambda_{pf1})t}$$

$$MTTF_1 = 1/(1+k)\lambda_{pfl} \qquad (1)$$

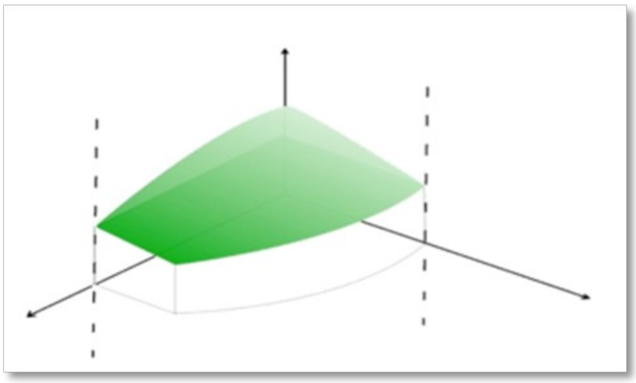

Fig. 2 Performance/reliability dependence

Fig. 3 presents a reliability degradation of the device without implementation of the fault detection and fault recovery). Introduction of the checking of hardware condition requires a hardware support.

In reliability terms, extra hardware has an extra rate of permanent and intermittent faults d λ pf1 and d λ if1 respectively, and reliability of the system with an implemented checking process is equal

$$P_{1+checking}(t) = e^{-((1+d)\ \lambda\ pf1+(1+d)\ \lambda\ if1)t} \tag{2}$$

where ***d*** is a share of the hardware redundancy required to implement the checking process. Redundancy for checking (detection) d varies in different hardware schemes. The maximum redundancy to provide checking is 100%, or d=1, when duplication of the hardware is used to compare outputs.

Maximum power of fault detection is a privilege of the duplication approach. Assume the same coefficient k of malfunction/permanent fault ratio

$$P_{1+checking}(t) = e^{-((1+d)+k)\lambda\ pf^*t}$$

$$MTTF_2 = 1/(1+d)+k)\lambda_{pf1} \tag{3}$$

Introduction of checking schemes makes the only difference in system reliability analysis by a guarantee that the appeared hardware fault (assuming the chosen fault type) is detectable. In other words, checking justifies reliability by transition of an observer from a random world into the world of conditional probabilities. Conditional probability describes a reliability of the system provided that fault of a chosen class did not happen.

So far, we assumed that our system has hardware to detect chosen and representative class of faults and that faults do not have latency period (a natural assumption for processors, cache memory and immune systems).

In contrast to detection, a recoverability of the system after intermittent faults requires more efforts and redundancy use, let's denote it r. Then the redundancy of hardware descends even more – the lowest curve of Fig 3.

$$P_3 = e^{-(1+d+r)(\lambda\ pf1)t} \tag{4}$$

$$\mathrm{MTTF}_3 = 1/(1+d+r)(1+k)\lambda_{pf1}t$$

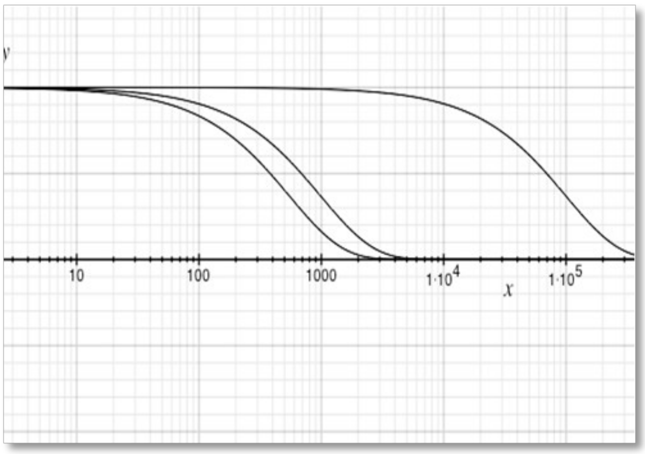

Fig. 3 R(t) with malfunction and checking

On the other hand, reliability might be increased, as long as malfunctions are tolerated. In the equations (4), a coefficient k of malfunction impact might be reduced, say, α times. Then

$$P_{1+checkingandrecovery} = e^{-(1+d+r)(1+\alpha k)(\lambda pf1)t} \tag{5}$$

$$\mathrm{MTTF}_4 = 1/(1+d+r)(1+\alpha\, k)\lambda_{pf1}t$$

The role of α defines a reliability gain that might be achieved by elimination of malfunctions. It is worth to analyze α a bit more.

B. Success Function

Let us define a success function of the malfunction reductions as SF. It is a well known fact that a double system covers all possible faults by comparison and while checking of the element is implemented it can recover from malfunctions [21]. Thus, a form of SF is defined by two values: initial and maximal. When no redundancy is used (x=0), a probability of recovery from malfunction is zero, SF→0. (Compare: if you don't pay for people healthcare then people will die from any disease…). In turn, using the known fact that 100% redundancy guarantees full success of system operation (including fault detection and complete recovery) one is able to write that SF →1.

A function SF=x*e1-x satisfies both initial conditions. Denoting coefficient alpha α as malfunctions reduction leads to:

$$\alpha = 1 - c^*(x^*e^{1-x}) \tag{6}$$

where ***c*** is a coverage of faults and directly depends on redundancy spent on detection c = f(d,λ) - the more we know the better, but at the cost.

Compare finally MTTF2 and MTTF1 (for the systems with malfunction elimination and without, respectively) calling it efficiency. Y axis denotes times of improvement, while X axis presents amount of redundancy involved:

$$E = (1+k)/[(1+d+r)(1+(1-c^*x^*e^{1-x})k)] \tag{7}$$

Figure 4 shows that higher coverage of fault and ability to tolerate malfunctions makes an element almost 3 times more efficient in comparison with a standard element. The gain will be much higher when a real malfunction to permanent fault ratio (100:1 or 1000:1) is applied.

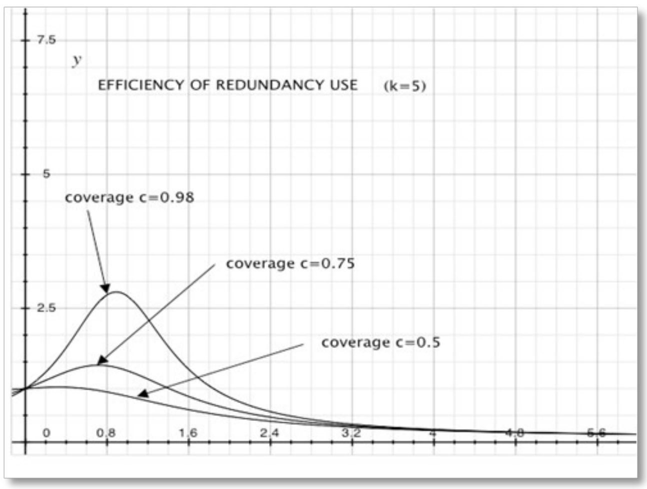

Fig. 4 Efficiency of redundancy

The estimation presented in Fig 4 does not separate use of redundancy between checking and recovery processes. This is a subject of further research; but the key outcomes are obvious and summarized below.

VI. Conclusion

Design of fault tolerant systems should be revisited: malfunctions must be tolerated at the element level, leaving permanent fault handling to the system level. There is an optimum in redundancy level spent on fault tolerance; this optimum is achieved with less than 100% redundancy. Efficiency of fault tolerance implementation in reliability terms depends on coverage of faults.

Acknowledgement

The author thanks Prof. Janusz Zalewski and Prof. Andrew Kornecki for encouragement, support and critique in the preparation of the paper.

References

[1] Schagaev I., Using information redundancy for program rollback, *Automatic and Remote Control* , pp. 1009-1017, July 1986.

[2] Sogomonyan E., Schagaev I. Hardware and software means for fault-tolerance of computer system. *IBID* , No. 2, pp. 3-53, 1988.

[3] von Neumann J., Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: *Automata Studies, Ann. of Math. Studies No. 34* (C. E. Shannon and J. McCarthy, eds.), pp. 43-49. Princeton Univ. Press, Princeton, New Jersey

[4] Schagaev I., Yet another approach to classification of redundancy, Proc. FTSD, Prague, Czechoslovakia, May 1989, pp. 485-490.

[5] Schagaev I, J. Zalewski, Redundancy classification for Fault Tolerant Computer Design, *Proc. 2001 IEEE Systems, Man, and Cybernetics Conf.*, Tucson, AZ, October 7-10, 2001, Vol. 5, pp. 3193-3198.

[6] Avizienis A., The Star (Self-Testing and Repairing) computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design, *IEEE Trans. on Computers*, Vol. C-20 (11), pp. 1312-1321.

[7] Avizienis A., Architectures of fault tolerant computing systems, *Proc. FTCS Symposium* , 1975, pp. 3-16.

[8] Avizienis A.,,.Ng Ying W, An Unified Reliability Model for Fault Tolerant Computers, *IEEE Trans. Computers* , Vol. C-29, No.11,.pp. 1002-1011, November 1980.

[9] Avizienis A., Laprie J., Dependable computing: from concepts to design diversity. *Proc. IEEE* , Vol. 74, No. 5, May 1986.

[10] Avizienis A., N-version approach to fault tolerance software. *IEEE Trans. on Software Eng ineering* , Vol. SE-11, No. 12, pp. 1491-1501, Dec. 1985.

[11] Laprie J.C., A. Costes, Dependability: a unifying computer for reliable computing, *Proc. 12th Int. Symp. on Fault tolerant computing systems, FTCS-12* , Los Angeles, June 1982, pp. 18-21.

[12] Laprie J.C., Dependable Computing and fault-tolerance: concepts and terminology, IFIP WG 104, Summer 1984 meeting, Kissimmee, Florida; *LAAS Research Report No. 84.035* , June 1984.

[13] Kopetz H. et al, Tolerating transient faults in MARS, *Proc. 20th Int'l. Symp. on Fault Tolerant Computing Systems* , Newcastle Upon Tyne, U.K., June 1990, pp. 466-473.

[14] Siewiorek D. Reliable *Computer Systems: Design and Evaluation.* Burlington, MA, Digital Press 1998.

[15] Stepanyants A., et al.. Malfunction Tolerant Processor and Its Reliability Analysis, *DSN 2001* , G ö teborg, Sweden.

[16] O'Brien F, Rollback point insertion strategies, *Proc. FCTS-6* , Pittsburgh, Penn., 138-142 (1978).

[17] Rendall B., System structure for software fault tolerance. *IEEE Trans. on Software Engineering* , .Vol. SE-1, No. 2, pp. 191-209, June 1975.

[18] Russell D.L, M.J. Tiedeman, Multiprocess recovery using conversations, Proc. FCTS-9, 1979, pp. 106-109.

[19] DeAngelis D,. Lauro J, Software recovery in the fault-tolerant spaceborne computer, Proc. FCTS-6, Pittsburgh, Penn., June 1976, p. 143.

[20] Wirth N., Gutknecht J.: *Project Oberon* . Addison-Wesley 1992.

[21] Schagaev I., Algorithms of Computation Recovery, . *Automatic and Remote Control*, 7, 1986.

[22] Schagaev I., Algorithms to Restoring a Computing Process, *Automatic and Remote Control,* 7, 1987.

[23] Schagaev I., Determination of type of hardware faults by software means. *Automatic and Remote Control,* 3, 1990.

[24] Vilenkin S., Schagaev I., Operating System for Fault Tolerant SIMD *Computers. Programmirovanie* , No.3, 1988 (In Russian).

[25] Pierce W.H., *Failure-Tolerant Computer Design* , Academic Press Inc New York, 1965.

[26] Birolini A., *Reliability Engineering Theory and Practice. 8e* , Springer, 2007.