Preprints of the 30th IFAC Workshop on Real-Time Programming and
4th International Workshop on Real-Time Software, pp. 79 – 84

Hardware Testing on the Level of Tasks

Thomas Kaegi -Trachsel*, Igor Schagaev**
Juerg Gutknecht*

**ETH Zurich, Zurich , Switzerland (e-mail: {thomas.kaegi, gutknecht}@inf.ethz.ch)*
***London Metropolitan University, London England (e-mail: i.schagaev@londonmet.ac.uk)*

Abstract: In this paper we give an introduction into fault tolerant computing and explain the generalized algorithm of fault tolerance with the several steps required to make a system fault tolerant. We show that having a hardware test assigned to every task is a powerful way to detect certain types of faults. To reduce the testing overhead and keep the system responsive, we show an algorithm that partitions the tests into the ones that are supposed to run asynchronously from the synchronous tests.

Keywords : real-time, scheduling algorithms, fault tolerance, operating systems, diagnostic tests

1. INTRODUCTION

Applications for fault tolerant systems are expanding faster than we could have imagined and are steadily applied in new fields, especially in embedded systems. Any embedded safety critical system, for example the ones used in air plane on-board systems must provide two features, namely Fault Tolerance (FT) and Real Time (RT) behaviour. For a system to provide these two features, the hardware (HW) as well as the system software (SSW) must tightly work together and therefore be designed and implemented in a mutual depended process. In addition, it is essential to already know the types of faults the system must tolerate (Sogomonyan et al., 1988; Siewiorek, 1998; Pierce, 1965).

In general, two ways exist how to create a reliable system. First by using and combining multiple unreliable, or in other words unhardened components (von Neumann, 1956) or second by using higher reliability components, hardened components or such that include various types of internal redundancy to maximise their reliability and efficiency. (Schagaev I., 1989; Zalewski et al. 2001)).

Over the years, various system types were defined that react to faults in different ways. A *graceful degradable system* is a system that can recover itself after the occurrence of a fault and continue processing in a degraded mode (Laprie, 1995). A system that can stop itself correctly after the detection of a fault is called a *fail-stop system* (Kopetz et al., 1990).

Consequently, Avizienis (Avizienis et al., 1971; Avizienis, 1975; Ying, Avizienis, 1980; Avizienis, Laprie, 1986), Laprie (Avizienis, Laprie, 1986; Laprie 1995) and Siewiorek (Siewiorek, 1998) defined that a system is *fault tolerant* if it recovers itself to full performance or at least to a degraded mode that still provides all essential functionality.

Faults are typically separated into two categories, *permanent faults* and *malfunctions*. Malfunctions are transient faults usually induced by external events such as alpha particle impacts. In contrast to malfunctions that do not permanently damage the hardware, it is not possible to fix permanent faults. These faults can have different manifestations such as stuck bits or an arbitrary behaviour of a component (Siewiorek, 1990). Practice shows, that malfunctions occur much more often than permanent faults, especially in space borne systems or flight systems, where the ratio of malfunctions to permanent faults can be up to $10^3 - 10^6$.

2. GENERALISED ALGORITHM OF FAULT TOLERANCE

Several authors (Sogomonyan et al., 1988; Zalewski et al., 2001) proposed to consider fault tolerance not as a feature, but as a *process* called *Generalized Algorithm of Fault Tolerance* (GAFT). Different types of redundancy (*information*, *time* and *structural)* are used for the implementation of GAFT, as shown in Table 1. Every step of the GAFT implementation is based on using various redundancy types either in software (SW) or in hardware (HW).

The taxonomy of Table 1 might be used for comparison and overview of different design solutions of FT systems. It also allows deriving the efficiency and fault coverage of every step in the algorithm. Therefore, it provides a tool to assist the developer in the selection of the most efficient solution in terms of overhead and coverage for the implementation of FT systems considering all possible approaches.

We define a system as *fault tolerant if and only if* it implements GAFT transparently for applications. Depending on the actual implementation, the algorithm may vary in the time required to execute each step, the used redundancy types and the tolerated fault types.

As an example, the main function of an on-board flight control system is the implementation of control algorithms. Let's call the primary function "process one" or P_1 according to (Stepanyants et al., 2001). Therefore, if the system ensures for the process P_1 full functionality and transparent applica-

tion recovery from a predefined set of faults in a given time frame, the system is *fault tolerant*.

Table 1: GAFT vs. redundancy types

Steps	Redundancy types					
	hw			*sw*		
	i	*s*	*t*	*i*	*s*	*t*
A.Prove the absence of faults, ELSE					1	
B.Determine type of fault					1	
C.If fault type is permanent THEN					1	
D.Reconfigure hardware		2			2	
E.Prove software consistency					-	
F.Locate faulty states				3		3
G.Recover software				3		3

Table 2: Redundancy types

Nr.	*Name*	*Redundancy type*	*Description*
1	Task checking	SW(I,S)	Hardware checking on the level of tasks.
2	HW re-conf.	SW(S), HW(S)	Disable and exclude the faulty element
3	Task restart	SW(t), SW(i)	Task restart

Table 1 and Table 2 show a possible minimal implementation of GAFT where Table 2 shows all applied redundancy measures. For step E, we always consider the task as affected by the fault and restart it accordingly. It also shows that a redundancy measure can cover multiple steps in GAFT, as for example the software based checking can also detect the fault type.

The implementation of the hardware checking (step A of GAFT) can be done on different levels with different timings:

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

The different implementations have different properties in terms of timing, fault coverage, types of faults that can be tolerated, power consumption, complexity and cost. It is therefore wise to combine several different checking schemes in one system. For example, it might be beneficial to protect the processor and memory by hardware schemes on the level of instructions (duplicated processors, triplicate memory) and use higher level schemes (procedure or module) for the other hardware components due to cost and power constraints. The implementation levels are not mutually exclusive, as for example the combination of hardware and software based checking can significantly improve fault coverage.

In general, the higher the implementation level the less hardware support is required, but with higher timing and software coding overhead.

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

The complexity of GAFT implementations also depend on the types of faults that have to be tolerated. Even though malfunctions occur an order of magnitude more often than permanent faults, it is necessary to implement special schemes for HW reconfigurability and recoverability to eliminate the impact of *permanent faults* on the system.

3. CHECKING PROCESS ON THE LEVEL OF TASKS

We present a possible implementation of the first step of the GAFT algorithm, namely the proof of the hardware integrity on the task level. As shown in the previous section, only certain types of faults can be tolerated on the instruction level with reasonable hardware overheads. Thus it seems wise to use task based checking, which we discuss here further.

Consider a sequence of tests and programs (tasks) denoted T and P in Figure 1. The initial test T starts at every operation cycle followed by the task and a re-execution of the test. The successful completion of this cycle guarantees the absence of permanent faults in hardware during the execution of P. Even if a permanent fault occurs during the execution of P, the second test detects this. The actual implementation of such a test can be done in software or hardware (firmware based) or both combined.

Figure 1: Sequence of hardware integrity through program execution

As an optimisation, the second test execution can be considered as the initial test of the next program execution as shown in Figure 2.

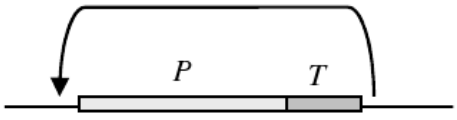

Figure 2: Regular sequence of program execution with proof of hardware integrity

During boot up, a testing phase is required to guarantee the correctness of the hardware before the first task execution. The applied tests might vary in depth (coverage), type of faults (single, multiple etc) and the set of the tested hardware.

Software based tests need a processor and memory to execute the test, even if a peripheral component is tested. In order to guarantee that faults in hardware components that are not subject of the test itself do not have an influence on the outcome of the test, the order of the tests must follow the *principle of growing core*: If a test of a hardware component u_i has implicit dependencies on another hardware component u_j, the test of u_j must be executed first.

If the resources needed by a task are known in advance, it is sufficient to run after the task execution only the testing procedures of the accessed hardware resources (*selective testing*), again by using the principle of growing core. This way, the system stays fully operational even if faults are present in hardware components that are not in use. Spare components could be used for relocation of a program running on a faulty hardware component. Of course, periodical tests of spare components are also required to ensure hardware integrity. If a non replaceable faulty hardware component is used by non critical tasks only, the system might also terminate these tasks and turn off the faulty component. This way, the system can still perform all critical tasks even in the case of faulty hardware.

For debugging and external diagnostic purposes, the results of the tests should be available for external hardware and within the system itself. Thus, the various hardware element test results should be organised in so-called *test syndromes*. For every hardware component, for example the register file, ALU, internal bus or cache memory state register, the checking procedures present the syndrome of the hardware checks, the system state in binary form as 0,0,0,0,0,0. In case of a non-zero syndrome, further analysis of the hardware conditions is required to define actions. This is not covered in this paper as it would correspond to step B to G in GAFT.

A test can also provide more information to ease recovery. If for example the testing schemes discover stuck bits in memory, it is sufficient to recover programs that access the affected locations and not all programs that are using this memory chip.

Hardware device drivers could provide their own testing schemes, possibly a combination of hardware and software based testing. I/O devices such as UARTs could effectively be tested by cross connecting the input and output wires by very simple additional hardware logic and transferring various character patterns in different modes.

3.1 Analysis of the checking process

Applications are nowadays so complex that they tend to saturate the computing system they are running on, which limits its diagnosis possibilities. Especially in multi processor systems, a high interest arises to test some hardware units when other hardware units execute tasks. This approach is called *sliding dropping diagn*osis (SDD) (Bogdanov, Schagaev, 1990).

In principle, two SDD types can be considered: *synchronous* and *asynchronous*.

An example: The CTSS Operating System for the CRAY-1 (Fong, Kirby, 1985) supports synchronous SDD with interruption of user tasks for testing hardware. Periodically every 15 minutes, the user tasks are interrupted and the diagnostic routines are executed. These routines are 'invisible' from the user task's point of view, as the entire system is stopped while the tests are executed.

The Synapse N+1 (Serlin, 1984) fault tolerant operating system serves as an example of asynchronous SDD. The OS implements a self loading algorithm to schedule tasks for processors. The same algorithm is applied for SDD too, as diagnostics routines are loaded as tasks and run on free processors.

The actions required for the implementation of synchronous SDD are as follows:

1. Unloading of the currently running task state from some hardware parts (e.g. RAM)
2. Loading and initialization of the diagnostic routines
3. Execution of the diagnostic process
4. Unloading of the diagnostic routines and performing further actions in case of faults. If high priority interrupts occur during testing, some temporary data might be needed to continue the testing after processing the interrupt
5. Reloading of the user task and continue processing

As asynchronous SDD tests do not interrupt running processes, as they are allocated on free processors, only steps 2 – 4 are required. The task unloading time is much higher than the task switch time, as the resources occupied by this task must be released, which might also involve moving the code and data of this application to another part of the memory. The same applies of course for reloading a task.

For the further analysis of the SDD, we consider the following three cases:

1) T_c and all task completion times are known.
2) T_c is known but the task completion times are not.
3) T_c is unlimited and the task completion times are not known

A SDD process that ends in checking all hardware units is called a *diagnostic cycle* and requires time T_c to complete.

3.2 The system model

The diagnosis algorithms and all analysis in this chapter are done on the basis of a multi processor system with a set of U identical processors (units). A single processor system is included in this analysis as it corresponds to the special case of $U = \{u_1\}$, which would of course extremely simplify the analysis. In addition, we assume that all scheduled tasks are independent, i.e. they have no time, control or information dependencies. Further, the task switch time (not to be mistaken with the task load and unload time in case of synchronous diagnosis) is so short in contrast to the task execution time that it can be safely ignored.

We define the required time for a single processor to complete the asynchronous SDD as T_{ad} and to complete the synchronous SDD as T_{sd}. As all processors are considered equal, all T_{ad} and T_{sd} are equal as well. Both diagnosis modes share the common task of performing the actual test, whereas the synchronous mode also involves unloading and reloading of the interrupted task.

Consequently:

$$T_{sd} = T_{ad} + (T_u + T_r) \quad (1)$$

where T_u is the time of unloading and T_r the time of reloading the user task in synchronous diagnostics.

In order to have an upper bound on the checking overhead at any given point in time, we propose here that at any given moment in time, the diagnostic process can run on at most one processor. Otherwise, by accident, a simultaneous testing of all processors would turn the system completely unresponsive.

In this spirit, it is also necessary to relax the strict order of the testing explained in chapter 3, and allow the testing to be done during task execution. The recovery in this case would involve higher cost as the outcome of the last task run cannot be trusted and must be repeated as well.

3.3. Diagnostic process algorithm

The real value of the asynchronous SDD depends on the sequence in which the units are diagnosed. The most natural way to appoint a diagnostic process for a unit is when the unit gets free. However, if a long running task is assigned to a single processer, it is not diagnosed for a long time. Therefore, in this special case, the synchronous SDD is preferable.

On the other hand, time critical tasks in a real time system should not be interrupted, and thus the asynchronous SDD should be applied. To decrease time overheads but still provide completed testing, a combination of both diagnosis modes seems to be ideal.

According to Blazewicz (Blazewicz *et al*., 2007), even the simple case of the Problem $P \| C_{max}$ where a set of independent tasks is scheduled on identical processors without preemption with the goal to minimise the schedule length is a problem of complexity NP-hard. Thus, our problem which imposes further restrictions on the scheduling algorithm and is therefore is also NP-hard is not perfectly solvable in a real system with a high number of tasks. The use of an approximation algorithm seems to be therefore the way to go.

Consider case 1) where T_c and all task completion times are known. Assume there exists an upper bound for T_c. We now add every unit u_i in $U=\{u_i \dots u_n\}$, $i=1 \dots n$ either to U_a $(u_i \in U_a)$ if the completion time t_i of the task running on unit u_i satisfies $t_i + T_{ad} \le T_c$ or to U_s $(u_i \in U_s)$ if $t_i + T_{ad} > T_c$. Obviously, it is impossible to test any unit in U_s asynchronously, so this step seems logical. But even if $U_s=\{\}$ it might not be possible to reliably test all units in U_a asynchronously. In fact, when a user task has finished on one unit while a testing process is running on another processor, the scheduler will assign a new task to the just released processor according to the constraint that only one diagnostic procedure can run concurrently. This case is triggered if the completion time of two units u_i and u_j are in the range of $t_j - t_i \le T_{ad}$. Even though both tasks finish before $T_c - T_{ad}$, there is not enough time left to test both in asynchronous mode. Therefore, the separation of units to two subset U_a and U_s is necessary but not sufficient.

We present below the procedure P that chooses a subset of units R for the synchronous SDD in accordance with T_c and the task completion times. The subset $R \in U_a$ formed by procedure P is tested synchronously as well as all units from subset U_s. $R \cup U_s$ is therefore tested synchronously, $U_a \setminus R$ asynchronously.

3.4. Diagnostic procedure P

First, we sort all units $u_i \in U$ in increasing order according to their completion time. This step is skipped in the algorithm as any standard sorting algorithm with $O(n \log n)$ can be used.

For case 1), it is heuristically checked if $t_{i+1} > t_i + T_{ad}$. If this is the case (case a. of Figure 4), no testing is ongoing when the task u_{i+1} ends, and therefore the testing process can immediately be started. This case corresponds to step 3

```
i=1,k=1,R={}, j=i+1;
1. if t_i <= T_c -T_ad then
  2. while t_j <= T_c -T_ad do
    3. if t_j > t_i + kT_ad then
      k=1; i=j;
    else
       4. if t_j < t_i + kT_ad - (T_u + T_r) then
        assign user task to u_j;
        set new completion time of u_j ;
         restore ordering;
      else
        5. k=k+1;
         if t_i + kT_ad > T_c then
          u_j ∈ R;
        end;
      end;
     end;
    j=j+1;
  end
end
if j < n then
  u_k ∈ R, k={j…n}
end;
```

Figure 3: Procedure P

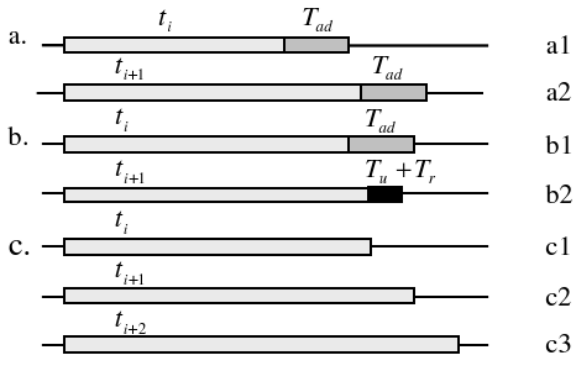

Figure 4: Task execution examples

of procedure P, when unit $u_j = u_{(i+1)}$ is left in the asynchronous testing mode.

Now, if this condition does not hold (case b. of Figure 4), the task $u_j = u_{i+1}$ finishes while the unit u_i is being diagnosed. Remember that the task u_j cannot finish before task u_i as all tasks are ordered according to their finishing time. If the task time including $T_u + T_r$ of this task is still smaller than $t_i + T_{ad}$ (case b. of Figure 4) then the idle time of this task is considered as too long or in other words the performance impact would be too high, and thus a new user task is assigned to u_j. Of course, the finishing time of u_j has to be adapted as well. In a next step, the just updated u_j must be placed at the correct position in all units to restore the ordering property.

For all yet undecided cases, i. e. $t_i + T_{ad} - (T_u + T_r) \le t_{i+1} < t_i + T_{ad}$, the decision is made easy: If enough time is available to perform the testing asynchronously, it is done so, otherwise synchronously (step 5 of Figure 3). Even if u_{i+1} is scheduled for synchronous testing and has to be included in set R, the time overheads will be less than with asynchronous testing but more waiting takes place.

We just illustrated the algorithm on the example of two tasks. The algorithm P in Figure 4 however was extended to support n units. It is also explained below why not the completion times for the tasks are used but an approximation instead.

A still unanswered question is when the synchronous diagnosis of $R \cup U_s$ should be performed. If possible, the tests for these units are executed within the gaps between the asynchronous diagnosis of the units $U_a \setminus R$, when the gaps are at least of length T_{sd}. If not enough gaps are available, the synchronous diagnosis could also be scheduled in gaps smaller than T_{sd}, which would delay the next T_{ad}. This approach is especially useful if the difference between the gap and T_{sd} is small. If T_c is exceeded, the problem is not schedulable and it is necessary to extend T_c.

If one wants to minimise performance losses when T_c is not limited, every unit except one gets a user task and the remaining one performs the asynchronous testing.

For case 2) where T_c is known but the task completion times are unknown, it is possible to perform the full testing if $T_c \ge n T_{sd}$. This ensures that even if all tasks are diagnosed in the more time consuming synchronous mode, enough time is available. For case 2) the testing algorithm is as follows:

Tests are performed asynchronously as long as enough time is left to perform all remaining tests synchronously, i.e. $(n-i)T_{sd} < (T_c - T^*)$ with T^* as the current time and i the number of the currently executed task.

In the case 3) where T_c is not known and therefore unlimited and the task completion times are also not known, it is hard to optimize the system diagnosis due to the lack of information. However, it is still possible to apply the here presented approach. Consider a system that currently tests unit u_i. Suppose now, that unit u_j completes a user task while u_i is tested. By using the test completion time which is known, it is possible to find out whether it is worth waiting for the test to complete or not. Thus, if the remaining testing time for u_i is less than $T_u + T_r$, it is worth for u_j to wait for the completion of u_i and then perform an asynchronous test. Of course, if u_j finishes and no testing is currently ongoing, u_j can immediately initiate its asynchronous testing. In other cases u_j has to be tested synchronously.

Although T_c cannot be strictly determined, the diagnosis process has a finite character as every task will finish eventually and thus every unit will be tested in the end.

3.5 Extension to diagnostic procedure

We now leverage the above constraint that all tasks are ready at $t = 0$ which is unrealistic especially in control systems and introduce a task ready time t_{ri} for every task. The task finishing time changes accordingly to $t_i = t_{ri} + t_{pi}$, i.e. the ready time + the task processing time of task number i.

In addition, we extend the task checking from the processor only to the full hardware used by the task and introduce t_{adi}, the asynchronous task checking time for task number i.

Taking this change into account, we modify the procedure in Figure 5. First, we want to keep the real time nature of the algorithm and sort the tasks according to their finishing time. Except for continuously running tasks, which can be tested at any time, it is advisable to move the testing to the end of a task processing to minimise recovery time in case of a fault.

```
i=1,R={}, j=i+1, t_sum =0;
If t_i <= T_c - t_adi then
```

```
  t_sum = t_adi ;
  while t_j <= T_c - t_adj do
    if t_j > t_i + t_sum then
      i=j; t_sum = t_adi ;
    else
       If  t_j < t_i + t_sum - (T_u + T_r ) then
        assign user task to u_j ,
        set new completion time of u_j,
         restore ordering
      else
        t_sum = t_sum + t_adj ;
         if t_i + t_sum > T_c then
          u_j ∈ R ;
         end;
      end;
    end;
    j=j+1;
  end
end
IF j < n THEN
   u  ∈ R, k={j…n};
  distribute(R);
END;
```

Figure 5: Modified algorithm P

The changes in the algorithm are twofold. First, the task checking time is no longer constant, thus we replace kT_{ad} by the sum of the task checking times t_{sum} .

Second, as not all tasks are ready at $t=0$, it is possible that some processors are free at some given time, which allows us to test the hardware used by tasks asynchronously even if they are not currently running. This is possible due to the nature of the applications in our case, namely reoccurring periodic tasks in a closed control system.

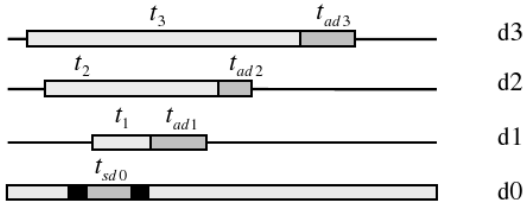

Figure 6: Example of new scheduling

Case d0 in Figure 6 shows the special case of a continuously running process. This task is synchronously tested at a time where no other processor is being tested. Task that perform only testing can run on all idle processors to increase fault coverage and reduce fault latency.

4. CONCLUSION & OUTLOOK

In this paper we showed the power of tasks with assigned hardware tests and show how they can efficiently be run on a multiprocessor system.

Further work on this topic includes the introduction of task dependencies, a constraint that is often given in real world applications, but not covered in this paper. Also helpful would be a thorough formal analysis of procedure P and its extensions.

The introduced algorithm is also applicable to time-sharing systems with preemptive multitasking. The preemption mechanism to schedule the tests might be helpful, however, this is subject to further research.

5. REFERENCES

Avizienis A. et al. (1971). 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

Avizienis A. (1975). Architectures of fault tolerant computing systems, *Proc. FTCS Symposium* , pp. 3-16

Avizienis A., Laprie J. (1986). Dependable computing: from concepts to design diversity. *Proc. IEEE*, Vol. 74, No. 5.

Blazewicz, J. et al. (2007). *Handbook on Scheduling, From Theory to Applications*, Springer Verlag Berlin Heidelberg, ISBN 978-3-540-28046-0

Bogdanov, J. J., Schagaev I. (1990), Sliding Slotting Diagnosys in Multiprocessors, *IMECO Congress Proc.* , pp. 141 – 150, Helsinki

Fong, Kirby W. (1985). The NMFECC Cray Time-Sharing System, *Softw., Pract. Exper.* , Vol. 15, pp. 87-103.

Laprie, J. C. (1995). Dependable computing and fault tolerance: concepts and terminology, *FTCS,* 1995, Highlights from Twenty-Five Years, Twenty-Fifth International Symposium on, pp. 2+.

Kopetz, H. et al. (1990). Tolerating transient faults in MARS, Proc., *20th International Symposium on Fault Tolerant Computing Systems* , Newcastle Upon Tyne, U.K., pp. 466-473

Pierce W.H. (1965), *Failure-Tolerant Computer Design*, Academic Press Inc., New York.

Schagaev I. (1989). Yet another approach to classification of redundancy, *Proc. FTSD* , Prague, Czechoslovakia, pp. 485-490

Serlin O. (1984). Fault-Tolerant Systems in Commercial Applications, *Computer* , vol. 17, no. 8, pp. 19-30,

Siewiorek D. (1990), Faults and Their Manifestation, *Fault-Tolerant Distributed Computing*, vol. 448/1990, pp. 244-261, Springer

Siewiorek D. (1998). Reliable Computer Systems: Design and Evaluation. Burlington, Digital Press

Stepanyants A., et al. (2001). Malfunction Tolerant Processor and Its Reliability Analysis, Goteborg, Sweden

Sogomonyan E., Schagaev I. et al. (1988). Hardware and software means for fault-tolerance of computer system. IBID, *Automatic and Remote Control* , No. 2, pp. 3-53.

von Neumann J. (1956). Probabilistic logic 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.

Ying Wang, Avizienis A. (1980). An Unified Reliability Model for Fault Tolerant Computers, IEEE Trans. Computers, Vol. C-29, No.11, pp. 1002-1011.

Zalewski J., Schagaev I. et al. (2001). Redundancy classification for Fault Tolerant Computer Design, Proc. 2001 IEEE Systems, Man, and Cybernetics Conf., Tucson, AZ, Vol. 5, pp. 3193-3198