Assessing Software Implemented Fault Detection and Fault Tolerance Mechanisms

P. Gawkowski, J. Sosnowski
Institute of Computer Science, Warsaw University of Technology, Warsaw, Poland,
email: gawkowsk@ii.pw.edu.pl, jss@ii.pw.edu.pl

Abstract

The problems of hardware fault detection and correction using software techniques are analysed. They relate to our experience with a large class of applications. The effectiveness of these techniques is studied using a special fault injector with various statistical tools. A new error-handling scheme is proposed.

1. Introduction

Dependability is a key issue in many applications ranging from simple embedded to complex mainframe or distributed systems. Various fault detection, diagnosis and error handling procedures have been developed. Software techniques are especially attractive when hardware design is fixed and cannot be modified (e.g. COTS elements, IP blocks in SoC design). An important issue is to check and improve the effectiveness of these techniques.

Software techniques for hardening system fault susceptibility are based on modification of the application code by introducing suitable procedures or instructions either at high-level source code or low level assembler code (e.g. [3,5,9,12,13,15,16]). Various fault injection experiments have been performed to demonstrate the effectiveness of these methods (e.g. [4,6] and references). Unfortunately most of them were limited to a few simple applications and the presented reports are not sufficiently detailed to reveal many problems appearing in practice (section 2 and 3).

We have analysed over 50 applications checked their natural susceptibility to faults and implemented various software fault detection and fault tolerance mechanisms (section 4). In our research we concentrate on the following problems: combining software procedures with available hardware error detectors, identification of weak points in software approaches, integration of error handling strategies. We have also studied fault susceptibility of the additional code in improved applications. This is especially crucial for frequent fault checking.

In the paper we show the impact of various program features which influence fault susceptibility (e.g. resource activity, compiler specificity, operational and test profiles). In the performed experiments we use software implemented fault injector FITS [9] which has been developed in our Institute. This injector delivers various original statistics and fault tracing procedures, which are not available in other reported tools.

2. Detection and tolerating faults

Hardware faults can be detected with the use of special on-line test mechanisms e.g. watchdog processors, control flow monitoring, error detection codes, invalid memory access or invalid instruction code detectors [17]. Some of them are available in COTS chips and systems. Typically faults are signalled by interrupt mechanisms (exceptions). Error confinement capabilities of these mechanisms can be enhanced at software level.

Software implemented fault hardening mechanisms are targeted for error detection or correction. Error detection is assured by simple assertions and control flow monitoring ([1,13]) or by various program modifications (replicating instructions, program variables, program modules and comparing results.). Fault tolerance or masking can be achieved with retries, error recovery, voting etc. These methods are especially effective in case of transient hardware faults. A large class of permanent faults can also be tolerated with RESO (recomputing with shifted operands), redundancy in RAM and CPU (e.g. superscalar structure) etc. [8, 15]. Moreover many data structures or files comprise some natural redundancy useful in detecting or correcting faults.

In fine-grained approach checking procedures are performed frequently e.g. every execution of duplicated instruction or a small piece of a program. In coarse-grained techniques this processes is initialized scarcely e.g. at the end of calculations. Fine granularity assures lower fault latency (at the cost of memory and time overhead). Quite efficient coarse-grained checking has been proposed in [14]. In this approach (called ED^{4}I) data diversity (and partially program) is assured by transforming an original program according to some rules (all variables and constants are multiplied by diversity factor k), performing computations with the original and transformed programs and comparison of results. Some kind

Proceedings of the 12th Asian Test Symposium (ATS'03)
1081-7735/03 $17.00 © 2003 **IEEE**

of fine-grained checking based on comparison was proposed in [3,16]. The authors developed rules of transforming a primary program into redundant structure by variable duplication (VDR) and control flow checking (CFCR) rules. Duplicated variables are compared on read operations. For error correction we can use additional rules (ECR) defined in [16]. They base on a checksum (used to identify the correct variable set) generated for one set of the duplicated variables. We have extended the original ECR rules to cover complex data structures (with pointers etc.).

Eliminating fault effects we face the problem of fault identification (transient, permanent and intermittent faults) and localization [2,17]. This leads to more complex fault handling procedures (discussed in the sequel).

3. Error handling problems

In the literature authors restrict their considerations to general ideas of fault detection or correction. Implementing these ideas in system environment for real applications we face two neglected problems: handling exceptions, managing error detection and recovery procedures. Built in hardware fault detectors (e.g. access violation, invalid opcode, memory misalignment errors) generate interrupts (exceptions) which are handled typically by the operating system. In a wide range of applications we found that such events contribute 10-60% depending upon fault localization (section 4). Hence even in the case of software procedures tolerating faults significant percentage of faults may not be corrected or masked due to generated exceptions. To overcome this problem we have to include exception handling at the application level. Some systems deliver appropriate mechanisms e.g. try and catch construct in object oriented languages of the following form:

```
try {segment of the application code}
catch (exception filter1) {specification of exception
 handling procedure}
.......
catch(exception filter n) {specification of exception
 handling procedure}
```

It assures taking over exceptions (specified by the filter or all of them – catch(...)) generated during the execution of the code within the brackets try{ ...}catch. For the specified exception we define appropriate handling procedure. For example it may initiate reexecution of the program code starting from some specified checkpoint (previously established – backward recovery), suspending further execution of the thread etc.

Handling exceptions we should notice that transient faults injected into registers and data memory can be recovered by retries or error recovery to previous checkpoint. More problems relate to faults in the program code. Even transient faults modify program code. They can be detected with the use of various checksums and then recovered. In general we distinguish several levels of recovery. If the fault appeared in the cache memory, then it can be recovered by flushing this memory (one or more levels of caches can be considered). A fault in RAM memory may need reloading from a stable memory (e.g. disc). To reduce recovery time several copies of the code segments can be stored in RAM and switched.

An important issue in fault handling is identification and localization of faults. Discrimination between transient, intermittent and permanent faults is based on collecting some statistics and performing autodiagnostic procedures. For this purpose we can use various algorithms [2,17]. They may lead to quite sophisticated fault handling concepts. For an illustration we present the developed scheme of coarse-grained fault detection and fault tolerance related to TMR (triple modular redundancy) system implemented in multithreaded environment. The main thread manages calculations in the processing threads and determines the final result.

```
void main(void)
{    InitializeInputData();
     CreateWorkingThreads(); // create processing threads
//they will be suspended waiting for data to compute
     while (Data_To_Process)   // while there is new
    // data to process
     {PrepareInputData();  // prepare input data
     StartToCompute();   // start (resume) computations
     //of all processing threads
     WaitForResults();  // wait for minimal number of
//results for the final decision
     SelectResults();   // determine reliable result
     PrepareNextIteration();} // prepare for the next
//iteration: and  if needed restart, terminate or suspend
//disturbed threads
}
```

In the presented code the comment lines are denoted with //. After some initialisation steps (initial input data preparation, creating processing threads) the main thread enters the main computation loop. Its iteration begins with preparing input data for processing threads. Then all processing threads receive the "green light" to begin computations. The main thread suspends itself waiting for results from processing threads. It is waked-up when at least 2 processing threads deliver results or the time-out occurs. This is assured by function WaitForResults. In Win32 such function can be realised with the use of synchronisation mechanism WaitForMultipleObjects. It returns signal if the specified wait criteria have been met: 1) timeout interval elapsed, 2) either one or all of the specified objects are in the signalled state. In the sequel we will use also simpler function WaitForSingleObjects, in which

criteria 2 is reduced to single object. In case of time-out situation, if any result is available it can be considered as the correct one (the result can be supplemented with some thread reliability specification) or a fixed safe result value might be delivered. Optional solution is retry or sending an error message to the user. If the main thread receives results from 2 threads it compares them. If they are consistent the result of the given iteration is selected. In this case there is no need to wait for the third thread computations (it can be cancelled and prepared for the next iteration). If two results are not consistent the main thread waits for the third result (or for time-out). Then the voting procedure determines the final iteration result. In more advanced fault handling some statistical information of the processing threads can be used (obtained from reliability monitoring in the main and processing threads). This statistics relates to the history of thread behaviour etc.

The processing threads may detect fault locally (e.g. by taking over exceptions) and perform error recovery procedure. This process can be enhanced by some statistical information delivered from the main thread (e.g. consistency of obtained result with selected correct one, fault statistics in previous iterations etc.). If the processing thread activity is not successful (e.g. infinite loop) than the main thread may terminate it, recreate it or reconfigure the system to the degraded structure (e.g. DMR). Here we give an outline of a processing thread.

```
Status WorkingThread(void)
{try{ CheckForConsistency(); }//testing the hardware
//and the thread
    catch(...) { return THREAD_INTERNAL_FAILURE; }
try{while(!finalize)  // the main computation loop
    {try{WaitForSingleObject(GreenLight); // suspend
//processing thread till the main thread synchronization
    MakeCheckpoint();// store initial state for error
    //recovery
    CheckForConsistency();
    Compute(InputData); // data processing
    ValidateResults(); // local checking of results
    ReturnResults(Result, ResultStatus);// signal
//availability of  results and its status to the main thread
    UpdateInternalStatistics(); }    //reliability statistics
    catch( InternalFailure )
    { return THREAD_INTERNAL_FAILURE;}
    catch(ComputationFailure || ...)
    {      // local recovery   from transient faults
    UpdateInternalStatistics();
try{  RollbackRecoveryFromCheckpoint();
    CheckForConsistency();Compute(InputData);
    VaildateResults();ReturnResults(Result,ResultStatus);
    UpdateInternalStatisticks();}
    catch(InternalFailure)
    {return THREAD_INTERNAL_FAILURE;}
    catch(ComputationFailure || ...) // unrecoverable
    //computation fault but consistent thread state
{ReturnResults(NOTHING,ResultStatus=INVALID);
    UpdateInternalStatistics();}} } }
    catch(...) { return THREAD_INTERNAL_FAILURE; }
return OK;}
```

The main thread is synchronized with the processing threads via GreenLight (starting iteration processing) and finalize (end of calculations) flags. The processing thread generates synchronization objects for the main thread in function ReturnResult. These objects are used by function WaitForResult in the main thread. The presented processing thread implements internal reliability monitoring and retry (rollback recovery with checkpointing) in case of fault detection (local error containment). The consistency check is performed after thread creation and for each processing iteration. Such frequent checking is useful only in critical applications. It detects faults during thread suspension. All functions specified in italic are optional. In general different schemes of error handling can be used in the main and the subordinate threads and optimized for application requirements, fault statistics etc. (compare [2]). We can move most of fault handling processing (in italic) to the main thread. Two types of exceptions are filtered in the processing thread: InternalFailure and ComputationFailure. The first one is more critical, it may relate to hardware permanent faults. The second one can be eliminated by error recovery.

In general error-handling process may be much more complex if we add the capability of distinguishing transient, intermittent and permanent faults. Here we can use algorithm proposed in [2] and [17]. Error detection, correction and fault handling procedures may contribute significant percentage of the final code and data area in the realised application. This is especially critical for frequently invoked procedures e.g. comparison, assertion checking, voting in fine-grained approaches. An important issue is to check fault susceptibility of these mechanisms and methods of hardening them (section 4).

4. Experimental results

Improving system fault robustness with software procedures we have verified their effectiveness in fault injection experiments with FITS. To get better insight into the problems we concentrate on three classes of experiments: checking natural fault susceptibility for a wide spectrum of applications, checking the effectiveness of fault hardened applications and checking fault susceptibility of fault handling procedures. Fault injection test results are qualified as correct (C – no fault impact on the result), incorrect (INC – not detected incorrect result), system exception (S) and time-out (T).

Having analysed over 50 applications we found that they differ in fault susceptibility. In tab.1 we give mini-

mal and maximal percentage of the obtained test results for single bit flips injected into CPU (Pentium) registers, data memory (used by the application) and code area.

Table 1. Distribution of test results (%)

Test result	Data	Code	Registers
Correct	1.92-76.2	3.36-16.7	27.5-61.0
Incorrect	11.3-97.5	23.8-57.0	4.13-37.0
System	0.0-15.0	16.1-66.2	13.1-52.0
time-out	0.0-2.43	0.1-12.0	0.0-1.02

Table 2. Register susceptibility to faults (%)

Applic.		EAX	EBX	ECX	EIP	ESP
LZW	C	95.6	100	96.2	9.1	26.5
	S	3.5	0.0	3.6	87.0	69.9
MIX1	C	60.1	82.7	92.3	3.8	15.8
	S	0.0	0.0	0.0	90.8	73.9

Quite big dispersion of fault effects for various applications relates to resource activity (the percentage of time it holds useful data) and code specificity [7]. Using various algorithms for the same problem we obtain different fault susceptibility (e.g. sorting algorithms with bubble, quicksort or selection assure correct results in the range 1.92-17.8%). It is worth noting that the same algorithm implemented by different programmers or with the use of different languages may differ significantly in fault susceptibility. Even more, the same source code translated with different compilers also gives different results [7] . In many applications the used input data set may influence program flow resulting in different fault susceptibility [6,7]. In general, applications strongly oriented for calculations are more susceptible to faults than applications dependant upon various relations.

In most publications authors give fault insertion results averaged over all injected faults e.g. [3,4,11,16]. Sometimes they give averaged figures for faults injected into code, data area or registers. It is worth noting that large scope of averaging may hide many real problems. Looking at results in tab.1 we can find big differences in test results depending upon the area of fault injection (code, registers, data area). Factors influencing test results have been outlined above. Tab. 2 shows register susceptibility to faults (C - percentage of correct results and S - system exceptions) for data compression algorithm LZW and a mixture of integer calculations (MIX1). Here it is worth noting that faults injected into some registers, strongly influencing program control flow (e.g. EIP, ESP, EBP), generate large percentage of system exceptions. This holds also for segment addressing registers [7]. Correlation of fault susceptibility and register activity we have analysed in [7].

Detailed knowledge on fault susceptibility facilitates optimisation of hardening procedures. On the other hand averaged figures could give some estimation of general fault susceptibility features. Correlation of these features with physical fault susceptibility is more complex. Some mapping can be found between experiment fault location distribution and chip areas. This may give better estimation of real fault susceptibility (compare [5]). However we should be conscious that FITS injector does not disturb directly some resources e.g. hidden registers (related to CPU microstructure and not accessible directly from the machine code), similar problem arises for complex combinational logic. So the mapping from physical fault distribution into RTL program level model needs further research (compare [5,10]).

In many applications some program segments are executed more frequently than others, so equal distribution of faults in memory address space may not well characterise the real fault susceptibility which mostly depends upon the most frequently used program segments. This situation is encountered especially in various embedded applications [9]. Hence it is important to explicitly specify what kind of fault distribution is used: (in space and time). Independent stressing (with faults) different program segments, registers etc. gives the possibility of deeper analysis of fault propagation etc.

In tab. 3 we present test results for applications with software implemented fault detection mechanisms. R1, R2 and R3 applications relate to Ar tgh calculation (Ar tgh = $0.5\ln 2\log_2(1+x)/(1-x)$ with $\log_2$ calculated according to Mitchell method), bubble sorting, and data compression according to LZW algorithm. All these applications were enhanced by modifying the primary programs according to fined grained approach: rules VDR (see section 2). Applications ED1 and ED2 relate to bubble sorting and Ar tgh calculations enhanced according to algorithm ED^{4}I [14] (section 2, coarse grained approach). AS is quick sort program with embedded assertions on correct table indices and final monotonicity of ordered output data.

Coarse grained error detection assures the lowest percentage of incorrect results. Fine-grained approach based on rules VDR is less effective for faults injected into code or registers (due to higher overhead of fault detection code – susceptible to faults). Quite high percentage of incorrect results (faults injected in data area) for applications R1 and R2 is caused by not checking the final result. This can be improved significantly by adding such checks (not included in the primary rules). Results for application AS show that simple assertion may not be effective. In all cases significant percentage of faults are detected by system exceptions (mostly faults injected into code and registers).

Analysing the effectiveness of fault detection mechanisms it is interesting to evaluate attenuation ratio of incorrect results A_{INC} = (incorrect results for the improved application)/(incorrect results for the primary application). For the considered applications we have obtained: A_{INC}(R1) = (0.23, 0.29, 0.65) , A_{INC}(R2) = (0.01, 0.01, 0.45) , A_{INC} (R3) = (0.91, 0.00, 0.00), A_{INC}(ED1) = (0.015, 0.00, 0.12), A_{INC} (ED2)=(0.30, 0.057, 0.00) , A_{INC} (AS) = (0.82, 0.53, 0.10). Figures in the brackets define attenuation ratio A_{INC} for faults injected into data, code and registers, respectively.

Table 3. Results for enhanced fault detection (%)

Appl.	Correct results			Incorrect results		
	Data	Code	Reg.	Data	Code	Reg.
R1	44.5	19.4	45.6	19.1	8.7	2.7
R2	78.86	17.9	45.6	0.12	2.86	6.7
R3	52.1	21.4	63.7	15.6	0.8	0.0
ED1	45.7	6.8	56.7	0.8	0.0	1.8
ED2	65.6	13	57.2	3.8	1.7	0.0
AS	23.9	18.8	60.9	54.1	8.4	0.9

Table 4. Results for fault tolerance mechanisms

Appl.	Correct results			Incorrect results		
	Data	Code	Reg.	Data	Code	Reg.
T1	100	82.4	78.9	0.0	0.4	1.0
T2	97.6	93.8	90.0	2.4	2.0	1.1
T3	99.7	98.2	93.0	0.0	0.0	0.0
RC	99.3	28.4	70.5	0.5	0.19	0.2
MA	35.4	4.7	31.1	0.5	0.0	0.2
MB	51.6	86.5	88.2	0.4	1.0	0.0

More correct results will be assured by embedding fault tolerance mechanisms. For illustration in tab. 4 we give results related to selected techniques:

- simple triple modular redundancy (TMR in code and data) based on final result voting i.e. coarse grained approach (applications T1 and T2 relate to CRC calculation and cryptography algorithm DES),
- more sophisticated TMR with three processing threads and retries according to the idea of section 3 (application T3 with CRC calculation)
- modification of the primary application according to error detection and error correction rules ECR described in section 2 (application RC is LZW data compressing application),
- application specific error correction. MA is matrix multiplication with checksums in rows and columns and error correction, application MB is enhanced MA by inclusion of code retry on detected errors.

All these techniques assure relatively low percentage of incorrect results and different fault tolerance capabilities (the remaining faults are detected). The highest percentage of correct results assures TMR coarse-grained redundancy. Nevertheless for short application T1 faults injected into code are tolerated only at 82.4% (due to relatively high contribution of voting code). Similarly for the fine grained fault tolerance technique used in RC application we obtained very low percentage of correct results for faults injected into code.

The best fault tolerance has been achieved in application T3 in which we applied efficient fault handling procedures described in section 3. In application MA faults are corrected using column and row checksums. This technique considered as efficient in hardware implementation does not increase fault tolerance capabilities if implemented in software (especially for faults injected into code). This can be improved with code retries initiated by detected faults (version MB). In all cases we observe some limitation on correct results for faults injected into registers. This is caused by faults injected into some registers (e.g. EIP, ESP, segment registers) generating system exceptions (special mechanism of fault handling in the system are needed to correct them).

Tab. 4 shows average ratio of correct results in code, data and register areas. More detailed analysis shows that different resources show different fault susceptibility. For an illustration faults injected into ECX, EDX, EDI registers resulted in 99.2-100% of correct results for two applications: data compression LZW hardened with ECR rules and a mixture of integer calculations (MIX1) with coarse grained TMR. However faults injected into EAX and ESI registers contributed 76.4% and 100% correct results (negligible incorrect results) for LZW application and 87.1% and 84.9% of correct results (12.7% and 15.1% of incorrect results) for MIX1 application. Faults injected into ESP, EBP and EIP registers are difficult for correction (they generate mostly system exceptions). Analysing the effectiveness of fault tolerance capabilities we should compare the ratio (FR) of correct results for the primary application and its fault hardened version. For the considered applications we have achieved: the following values of fault tolerance ratio FR(A,L) (where A specifies the application, L specifies fault location: D, C and R for data, code and registers):

FR(T1,D) = 2.42, FR(T1,C) = 24.5, FR(T1,R) = 2.87;
FR(T2,D) = 1.4, FR(T2,C) = 13.43, FR(T2,R) = 2.66;
FR(T3,D) = 2.35, FR(T3,C) = 59.16, FR(T3,R) = 2.66;
FR(RC,D) = 1.21, FR(RC,C) = 6.05, FR(RC,R) = 1.07
FR(MA,D) = 1.12, FR(MA,C) = 0.8, FR(MA,R)= 0.92;
FR(MB,D)= 1.64, FR(MB,C) = 6.05, FR(MB,R) = 2.

It is important to note that in general fine-grained fault tolerance mechanisms due to relatively high code overhead related to voting or other fault handling procedures show lower fault correction capabilities than coarse-grained approaches. For MIX1 application with fine grained TMR we achieved 92.5%, 57.9% and 71.3% of correct results for faults injected into data, code and regis-

ter area. For coarse-grained TMR these figure have increased to 95.6%, 82.2% and 82.7%.

Fault handling procedures are also prone to faults. Injecting faults into register and code of a voting procedure we have got correct results in the range 65-71% and 54-57% (for various voted data), respectively. Hence it is reasonable to make these procedure more robust (e.g. with redundancy [9]). We have also observed some problems with detecting disturbed sates of registers ESP. EBP and EIP stored on the program stack. This can be resolved by duplication.

5. Conclusion

In the paper we have shown that the effectiveness of COTS error detection mechanisms is limited and it depends upon various application features: code specificity, used algorithms, compilers etc. This effectiveness can be improved with software techniques (significant percentage of faults can be corrected). We have proposed original fault handling schemes integrating application and operating system level mechanisms.

Performing complex fault hardening scenarios (e.g. taking into account timeouts, multithreaded or replicated processing, retries etc.) we face the problem of event synchronization. Detailed analysis of fault injection results facilitates optimisation of fault hardening procedures. Some additional hardware detectors e.g. watchdog timers or more sophisticated watchdog processors can supplement software approaches.

References

1. A. Benso, S. Di Carlo, G. Di Nartale, P. Prrinetto, L. Tagliaferri, Control flow checking via regular expressions, *Proc. of the 10th Asian Test Symposium*, 2001, pp.299-303.
2. A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni, Threshold-based mechanisms to discriminate transient from intermittent faults, *IEEE Transactions on Computers*, Vol. 49, No. 3, March 2000, pp. 230-245.
3. G.C. Carderilli, F. Kaddur, A. Leanori, M. Ottavi, S. Pontarelli, R. Velzaco, Bit flip injection in processor based architectures: a case study, *Proc. of 6th IEEE On-Line Testing Workshop*, 2002, pp.117-128.
4. M. Chen, T.K. Tsai, R.K. Iyer, Fault injections and tools, *IEEE Computer*, April, 1997,pp.75-56.
5. P. Cheynet, et al., Experimentally evaluating an automatic approach for generating safety critical software with respect to transient errors, *IEEE Transactions on Nuclear Science*, vol.47, no.6, Dec. 2000, pp.231-236
6. Folkesson,, Karlsson, Considering workload input variations in error coverage considerations, *Proc. of EDCC-2*, Springer, 1999, pp.171-188.
7. P. Gawkowski, J. Sosnowski, Analyzing fault effects in fault insertion experiments, *Proc. of 6th IEEE On-Line Testing Workshop*, 2001, pp.21-24.
8. P. Gawkowski, J. Sosnowski, Experimental evaluation of fault handling mechanisms, *Proc. of 20th Int. Conf. SAFECOMP*, Springer Verlag, 2001, pp.109-118.
9. P. Gawkowski, J. Sosnowski, Using software implemented fault inserter in dependability analysis, *Proc. of IEEE Int. Symposium on Dependable Computing* ,.2002, pp. 81-88.
10. S. Kim, A. K. Somani,, Soft Error Sensitivity Characterisation for Microprocessor Dependability Enhancement Strategy, *Proc.. of IEEE Int. Conference on Dependable Systems and Networks* 2002, pp.416-428.
11. H. Madeira, R.R. Some, F.D. Costa, D. Rennels, Experimental evaluation of a COTS system for space applications, *Proc. of IEEE Int. Conference on Dependable Systems and Networks*, 2002, pp.325-330.
12. B. Nicolescu, R. Velazco, M.S. Reorda, Efectiveness and limitations of various software techniques for soft error detection, a comparative study, *Proc. of 7th IEEE Int. Testing Workshop*, 2001, pp.172-177.
13. N.Oh, P.P. Sivani, E.J. McCluskey, Control flow checking by software signature, *IEEE Transactions on Reliability*, vol.51,No.1, March 2002, 111-122.
14. N. Oh, S. Mitra, Mc J. Cluskey, ED4I Error detection by diverse data and duplicated instructions, *IEEE Transactions on Computers*, vol.51, No.2, Feb. 2002, pp.180-199.
15. N.Oh, P.P. Shrivani, E.J. McCluskey, Error detection by duplicated instructions in super scalar processors, *IEEE Trans. on Reliability*, vol. 51, March 2002, pp.63-75.
16. M. Rebaudengo, M.S.Reorda, M. Violante, A new software based technique for low cost fault tolerant application, *Proc. of An. Reliability and Maintainability Symp.*, 2003.
17. J. Sosnowski, Transient fault tolerance in digital systems, *IEEE Micro*, February 1994, pp.24-35.

Acknowledgement. This work was supported by KBN grant 4T11C049 25 in 2003.