Stack processor architecture and development methods suitable for dependable applications.

Conference Paper · January 2007

Source: DBLP

CITATIONS
9

READS
811

3 authors, including:

Camille Diou
University of Lorraine
48 PUBLICATIONS **192** CITATIONS
SEE PROFILE

Fabrice Monteiro
Université de Lorraine, Metz, France
104 PUBLICATIONS **599** CITATIONS
SEE PROFILE

Stack Processor Architecture and Development Methods Suitable for Dependable Applications

Mehdi JALLOULI [†], Camille DIOU [†], Fabrice MONTEIRO [‡], Abbas DANDACHE [‡]
[†, ‡] *: Laboratoire Interfaces, Capteurs et Microélectronique, Université Paul Verlaine*
7 rue Marconi, 57070 Metz France
[†] {nom@univ-metz.fr}, [‡]{prénom.nom@ieee.org}

Abstract

Nowadays, reconfigurable and multiprocessor systems are becoming increasingly attractive for many applications. Such systems should be more and more dependable especially if errors occur on bits which change the circuit functionality (reconfiguration bits). In addition, mechatronic and automatically controlled systems often work in harsh environmental conditions which make them more prone to errors due to various disturbances. Thus, designers must consider ways to protect them against such errors. In this paper, a special interest is allowed to the processor architecture dependability. First, we present a stack processor architecture suitable for dependable mechatronic applications (automation, process control) which is implemented using a stack processor emulator. Then, we show the results obtained using a protection method depending on fault injection models integrated on the emulator.

1. Introduction

The results presented here are a part of a Ph.D thesis currently under work at the LICM Laboratory of Metz (France). This work is a part of a large CETIM project whose principal objective is to define an integrated design of dependable mechatronic systems. The goal of this work is to propose and validate a design methodology of a dependable and easily configurable processor architecture.

A mechatronic system is generally composed of the processor, sensors, actuators and the application. To obtain complete dependable system, the dependability of its elements must be taken into account since the very beginning of the design process. Hardware techniques could be used to ensure dependability of a processor and other hardware components whereas software techniques are used to ensure the application dependability. The interfaces and transmission lines between the processor and the application and between the processor, sensors and actuators should also be protected. In summary, our work concerns principally the processor architecture dependability. Than, it will be integrated in a multiprocessor and reconfigurable environment. So many dependability factors should be taken into account [1], [2] in particular for mechatronic applications [3], [4].

The design methodology proposed here, as shown in Fig. 1, consists on developing a processor emulator integrating the results obtained from the constraints of a dependable mechatronic application. Some of the most important issues are:

- Practical context (realization of a demonstrator),
- Faults and errors modeling,
- Faults and errors protection methods,
- Exchange protocol between sensors, processors, actuators and users, and eventually communication and parallelism in case of reconfigurable multiprocessor system,
- Low cost,
- Time constraint, and
- Instruction set functional requirements.

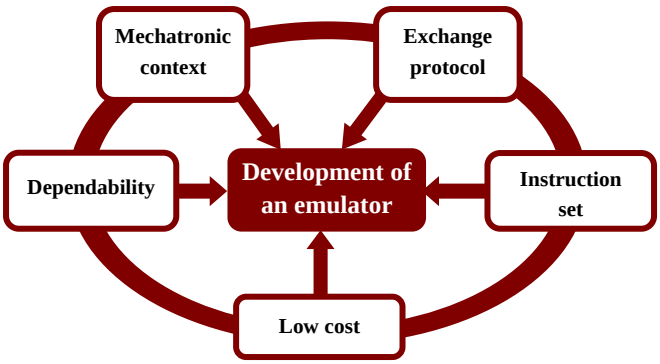


Figure 1. Design methodology

A particular interest is dedicated to these two features: the dependability and the mechatronic application, through an identification of faults and errors which can be produced in a mechatronic environment in addition to the appropriate and adapted fault and error protection techniques. Indeed, one of the primary objectives is to have an evolutionary

emulator in which we could integrate errors and faults models and benchmarks in which we could integrate their protection methods. So, the emulator must be the principal tool of the architecture fine-tuning. However, before developing the emulator, we must establish some choices concerning the most appropriate architecture. The first part of this paper introduces the stack processor architecture and the reasons of its choice. The second part details the internal structure of the stack processor adopted. The third part presents the emulator. And the final part shows the time performance of a software protection method against errors models implemented in the emulator which be followed by conclusions and future works.

2. Why stack processor ?

Before developing the emulator, we must establish some choices concerning the most appropriate architecture like CISC, RISC and MISC (Minimal Instruction Set Computer). In fact, their classification is based on different criteria such as cost, performance, instruction set, capacity (memory size, data word length and size of the secondary storage), component base and others [5]. Comparing to RISC architecture, the instruction set in MISC is further minimized, resulting in a low cost processor with reasonably high performance, like the M17 microprocessor [6]. Following the comparison between CISC, RISC and MISC architectures done in [5] and our design paradigm for dependability aiming for research is simplicity, we have chosen the MISC architecture because of its advantages (smallest area, few instruction count and fixed instruction length). In addition, concerning dependability features, we note that duplication is low cost and the core protection is less complicated on MISC than on the other processors. Amongst MISC processors, the stack processor is chosen because of its simplicity and flexibility. In fact, from a theorical viewpoint, stacks themselves are important, since stacks are the most basic and natural tool that can be used in processing well structured code. Stack machines are easier to write compilers for. And since we are designing an adaptable processor, building a machine that can have an efficient compiler is important [6]. Some recent applications are also using stack processors [7], [8].

Before detailing the stack processor architecture, we shall explore the stack computer design space which is well detailed in [6]. It may be categorized by coordinates along a three-axis system as shown in Fig. 2: the number of stacks supported by hardware (single or multiple), the size of any dedicated buffer for stack elements (small or large), and the number of operands permitted by the instruction format (0, 1 or 2). To adapt the stack architecture according to the application needs, we must have an initial stack machine model which is called the canonical stack machine [6]. It contains data bus, data stack and return stack with their top-of-stack register, arithmetic/logic unit, program counter, program memory with memory address register, control logic with instruction register, and input/output section.

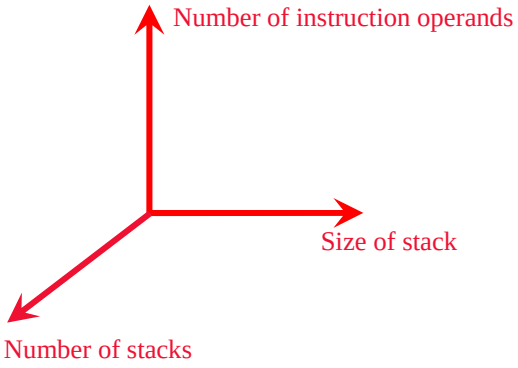


Figure 2. The three-axis stack design

3. Stack processor architecture

Now, we will specify the proposed architecture. First of all, to note that the target application domain is mechatronic, automation, and process control, i.e. it is neither pure calculation nor signal processing. Therefore, we have formulated some choices concerning the three dimensions of the stack design. For the number of stacks, we should notice that there is no universal single stack machine without auxiliary registers. To satisfy the simplicity requirement, we opted for a 2-stack processor. Concerning the size of stack buffers, we have chosen a large stack buffers which allow for multiple storage of data without loss. Their other advantage is that program memory cycles are not consumed while accessing data elements and subroutine return addresses, which can lead to significant speedups, particularly in subroutine-intensive environments. Concerning the number of operands, we have chosen a 0-operand instruction because all specified operations are performed on top-of-stack elements (except pushing data). So, we don't need to associate an operand to the operation code and in addition, this will reduce the instruction length. We use the 0-operand 2-large stacks processor. Concerning data bus and instruction length, 16-bit data bus is sufficient because of two reasons. Firstly, with an 8-bit architecture, execution of any 16-bit instruction takes more time (dividing the data into low and high, executing instruction twice, and putting together two results). Secondly, the data comes from the sensor through an A/D converter, so no accuracy more than 16 bits of an A/D conversion is needed. As for the instruction length, an 8-bit instruction is sufficient, because of two reasons. Firstly, in a zero-operand stack processor, the number of instructions is limited (<256). Secondly, 8 bits correspond to the minimal size of 2^i. Finally, an external stack is chosen, because it allows a large stack size to be pointed by a pointer (non addressed register).

For the above architecture, the following issues are addressed now:

- Formulation of the instruction set requirements,
- Construction of the data path,
- Development of the emulator, and
- Test and simulation of all instructions.

Some 34 8-bit instructions are proposed, including those for manipulating data and return stacks elements, classical arithmetic, logic instructions and addressing instructions. For the above set of instructions, the complete data path is proposed.

4. Stack processor emulator

In order to verify the functioning of the architecture proposed and to describe the functional behaviour of the processor and its internal states, we have developed the 2-stack processor emulator. The input of the emulator is a classical hexadecimal file (benchmark) which characterizes the application as it is shown in Fig. 3. It is developed in C language and contains, at the moment, about 2000 code lines.

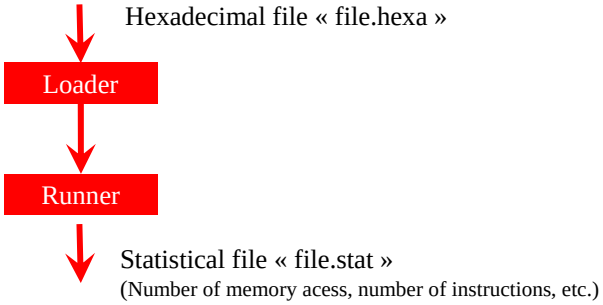


Figure 3. Emulator structure

To run an application, the program is executed and the emulator modifies data into memory, stacks and pointers, according to instructions specified in the hexadecimal input file. In addition, the emulator gathers statistics such as determining the number of each memory accesses and the account of each instruction. So, the emulator evaluates the internal states and the running duration. Eventually, the benchmark corresponds to the processor embedded programs and it is tested in the emulator. Firstly, the tests give statistics on different memory access, most frequently instructions, cycle numbers depending on different memory strategies (2 or 3 bus, we will discuss later). These statistics will have consequences on the architecture. Secondly, some functional errors can be modelled and integrated in the emulator. As a consequence, the program and the internal processor states may change. In order to skip such case, some protection techniques can be added in the benchmarks as it is shown in Fig. 4. This is very useful because our priority is the dependability.

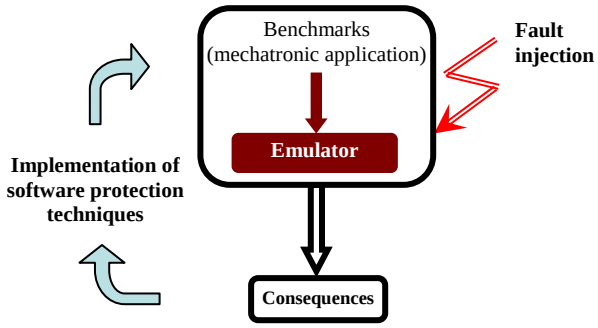


Figure 4. Usefulness of the emulator/benchmark for fault injection and protection methods integration

5. Experiments and results

5.1. Effects on adress bus

Concerning the number of processor address bus, the processor can be addressed through 2 or 3 buses. In fact, the instructions which are in the program memory (M1) treat 3 others memory areas: the data stack memory (M2), the return stack memory (M3) and the explicit memory (M4). M3 allows the storage of the function address in case of function calls and also the storage of data in case of temporary copy between the two stacks. M4 contains data addressed directly.

Two addressing bus strategies are possible as it shown in Fig. 5.

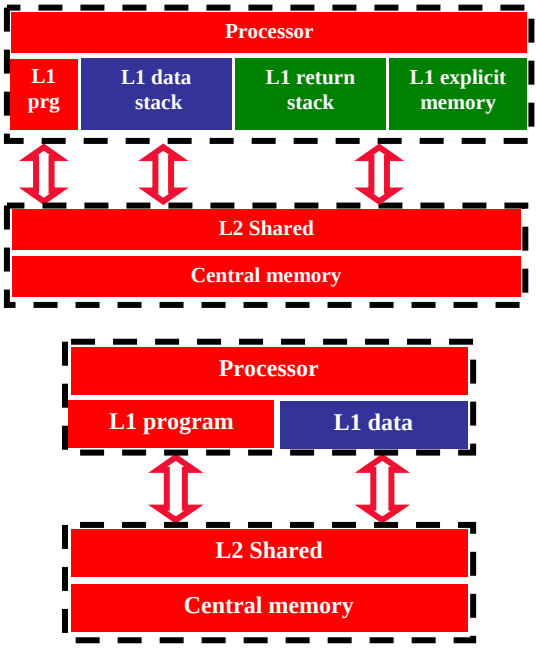


Figure 5. 3-bus and 2-bus strategy

We compare the time performance of the two strategies using the emulator and benchmarks. For the moment, 3 different benchmarks are developed:

- Benchmark which computes logic and arithmetic equations,
- Benchmark which permutates two by two ten data stored in memory, and
- Benchmark which sorts 10 data with 5 different initial conditions (IC1… IC5).

This comparison is illustrated in the Table 1 below:

Table 1. Time performance loss of the 2-bus strategy

	Time performance loss 2 bus comparing 3 bus
Benchmark 1	**31%**
Benchmark 2	**35%**
Benchmark 3 (IC1)	**30%**
Benchmark 3 (IC 2)	**28%**
Benchmark 3 (IC 3)	**28%**
Benchmark 3 (IC 4)	**27%**
Benchmark 3 (IC 5)	**28%**
Average	**30%**

As we can see, with 2-bus strategy, the program duration is 30% longer. This loss can have a consequence on dependability. In fact, if we suppose that the program duration is 6 ms in 3-bus strategy (8 ms in 2-bus strategy because of the 30%), and that the processor is dependable for the rate of 1 error every 7 ms, in this case the 3-bus strategy is better. In addition to this time performance loss, there is a risk to lose other time performance due to the eventual use of software protection techniques. In fact, these techniques can extend the program duration because of the protection routines which may be integrated in benchmarks. To avoid this program extension, hardware protection techniques can be used. So a compromise should be found between hardware and software protection.

5.2. Implementation of a protection method and its consequence on dependability

The protection techniques consist of a share between hardware techniques and software techniques. Here, we interest to errors which can be produced in the processor and not in the application. If we look at the consequences of these techniques, we notice that on one hand, when implementing hardware techniques, we will have an additional silicon area and possibly extend the critical path. On the other hand, when implementing software techniques, we will have a loss on time performance because of the additional protection routines. So, for an application located in particular conditions (error rate, time constraints, power constraints), we can estimate which kind of protection should we use. Thus, a compromise should be found. In summary, each protection technique has its efficiency and cost [9]. For that reason, we use both emulator and benchmark to determine the limit between software and hardware protections. In the next paragraph, we present a software protection technique, its possibilities to correct errors and its impacts on time performance depending on errors injection models.

The errors injection is implemented in the emulator. We suppose the existence of mechanism of hardware detection technique. Every detected and not corrected error generates an interruption. This interruption forces the processor to run a routine from a definite address. The function of this routine is to return to the last dependable state. In fact, in the benchmark, we integrate a routine which store periodically the stack pointers, the program counter and the top-of-stacks. Storage of the treated data is also possible. So, on every interruption, the protection routine reads the last data saved. These 2 routines (storing data and reading stored data) are implemented in the sort benchmark.

Our objective is to have an idea about the time consequences of this technique depending on the first moment of the errors apparition, its duration and its frequency. Some scenarios of error apparition are supposed and implemented in the emulator. Here, we present the 4 adopted models in Fig. 6.

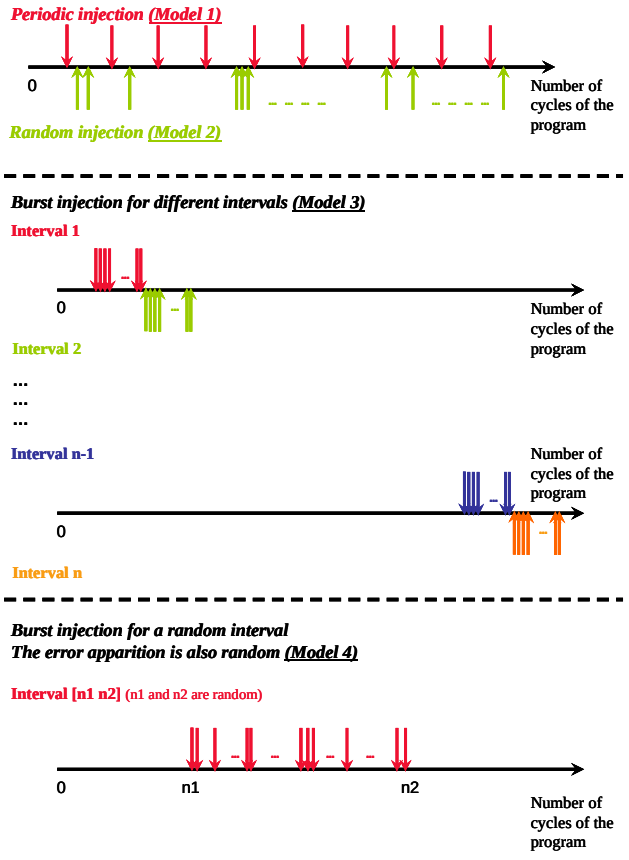


Figure 6. The error models adopted

For the first model, we produce an interruption every N instructions. For the second model, the error apparition is aleatory. For the third model, we choose to produce 10, 20 or 40 errors during a definite interval. This test is repeated for 9 possible intervals which duration is 200 cycles. Finally, for the forth model, the interval and the number of error injected are aleatory. Every model is tested with two possible protection methods: the first which consist on the storage of the stack pointers, the program counter and the top-of-stacks (partial storage) and the second which consist on the storage of these parameters in addition to the sorted data (complete storage).

As a first result, in the table 2, we have noticed the time performance loss due to the 2 bus strategy and also due to storage and protection.

Table 2. Time consequences due to storage protection and due to 2-bus strategy

	Count of cycles		Additional time needed for the 2-bus strategy (%)	Protection additional time (%)	
	3 bus	2 bus		3 bus	2 bus
Benchmark without protection	1833	2333	27%		
Benchmark with periodical storage	2367	2987	26%	29%	28%
Benchmark with periodical storage and protection every error (1 error / 250 instructions)	2694	3455	28%	47%	48%

This technique requires 48% additional time to correct 10 errors (1 error every 250 instructions). Even with protection, the 2-bus strategy program duration is 27% longer (near the 30% average found before).

This is the case of 1error every 250 instructions. We propose to change the periodicity of the error injection in order to find the maximum error rate that could be corrected. So, the error injection is implemented as we have described the first model.

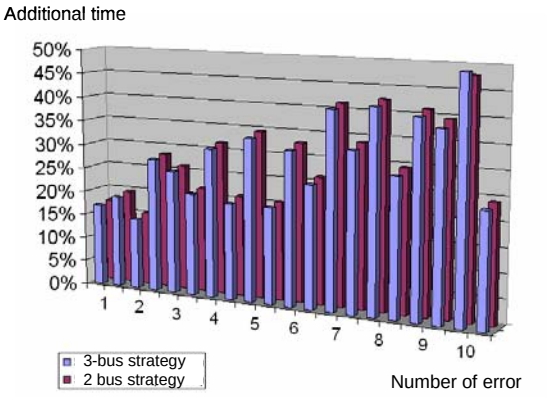


Figure 7. The additional time loss due to error correction (model 1)

As we can see in the Fig. 7, we can not correct more than 10 errors by program in case of partial storage. We have tested the case of complete storage and we have found 9 possible correctable errors. For the partial storage, the maximum rate of error that could be corrected is 1 error every 207 cycles. For more errors, the program spends time making a saving and a return to the last correct state (because the error frequency is higher). In that case, it is necessary to find hardware techniques (or the other software techniques) to have a protection for a higher rate of errors.

Now, we change the way of error injection and make it unpredictable. We look at the results in this Fig. 8.

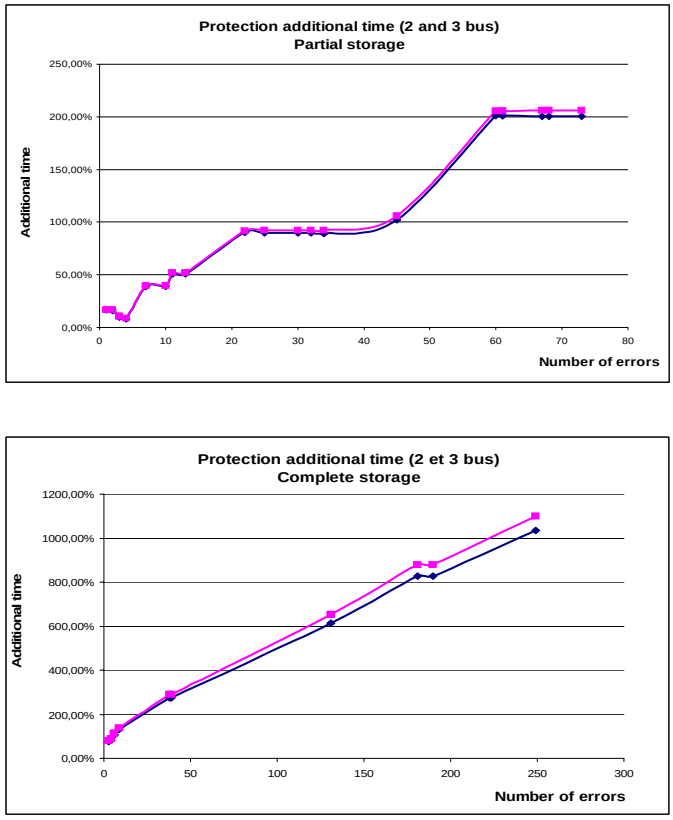
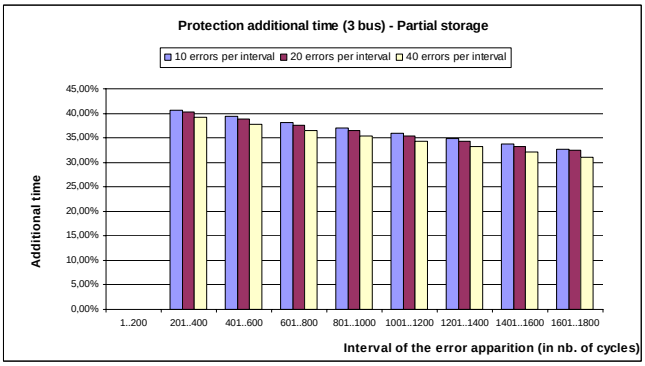


Figure 8. The additional time loss due to error correction (model 2)

By changing the way of injection of errors and making it unpredictable, we achieve to correct 73 errors with 200 % of additional cost (this without the saving of the sorted out data). That is, we divide the frequency by three which is not insignificant all the same.

By taking into account in the saving of the context, the handled data (besides 10 data to be sorted out), we manage to correct 249 errors. However, the temporal additional cost for this maximum of errors that could be corrected is 1000 % which is huge. An acceptable maximum additional cost of 130 % allows us to correct 9 errors only. Beyond, we lose a lot in temporal performances.

This time, we suggest injecting errors in a frequent way (model 3) or randomly (model 4) during a definite interval. This interval can also be defined randomly. For the model 3, as far as the normal execution of the program of unprotected sorting lasts 1833 cycles (3 buses), we then choose to inject errors during intervals of 200 cycles by changing it at the beginning of the interval of 0 to 1600. We raise the results according to the frequency of the errors (every 5 cycles, 10 cycles or 20 cycles). Obviously, other tests could be done by changing the appearance moment of the first error as well as the duration of the burst. The obtained results are illustrated in the Fig. 9.



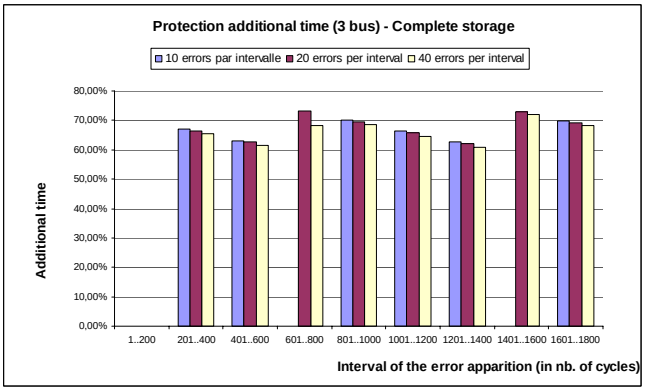
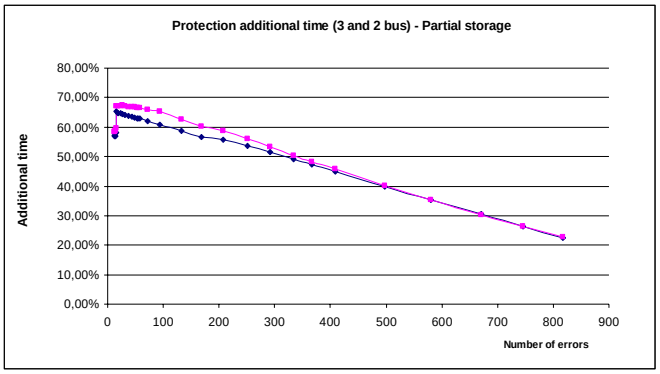


Figure 9. The additional time loss due to error correction (model 3)

What is interesting to note, it is that for errors which appear from the beginning, the technique can not correct them. This is due to the fact that the first saving of the context is done after 32 cycles. Thus, an error which appears before 32 cycles should be corrected otherwise. The most judicious would be, for example, to add a function which makes a return to the initial state.

The second observation concerns the difference of the temporal additional costs for a given interval. Indeed, the execution of the program with a correction of 10 errors lasts a little more than the one with a correction of 40 errors (a additional duration from 1 to 2 % for all the studied intervals and this for both cases partial and complete storage). This seems a priori illogical because in theory the correction of 40 errors, that is the execution of 40 times the routine of reading the last stored data, needs more cycles than its execution during 10 times. However, what takes place is that each of these 39 returns to the last safe state do not finish their routines before the apparition of an other error and a demand of re-execution of this routine. That means that only the last routine is completely executed.

This time, we suggest injecting errors in a random way (model 4) during an interval chosen also in a random way. This interval is the following one: [722 1540] cycles. The superior border does not exceed 1833 cycles as far as the normal execution of the program of sorting without protection lasts 1833 cycles (by 3 buses). We then chose to inject errors randomly during this interval and that, on an algorithmic point of view, the numbers pulled randomly are lower than a certain threshold. The variation of this threshold allows us to have various possible numbers of errors. We so raise the results of additional costs according to this number of errors, in other words according to the duration of the burst errors. The obtained results are illustrated in the Fig. 9.



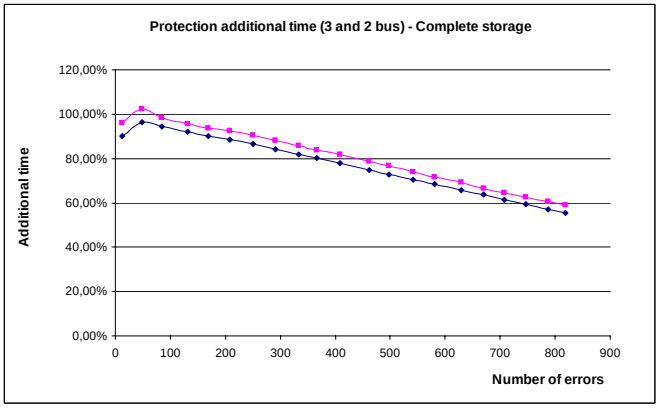


Figure 10. The additional time loss due to error correction (model 4)

We notice that the protection additional time is lower than 70 % in case of partial storage, and lower than 103 % in case of complete storage. What's interesting is that if the frequencies of the errors are important, the routines of protection to these errors do not run until its ends. While with less important errors frequency, the protection routines of these errors are altogether executed (the logical explication of the diminution of the additional cost).

As a synthesis, we notice that with periodic injection of errors, the absence of the growth of the additional cost is distorted by the cyclic phenomenon of the errors. With random errors injection, additional time begins to be penalizing and more comparing to the additional time of burst injection. The intention to find a hardware / software compromise is confirmed because of the excessive additional time cost from a certain number of errors. Whereas for burst injection, the errors are not also disturbing as those which appear in a unpredictable way. The additional cost of an important number of errors which appears during a short duration is less expensive than when the same number of errors appears randomly. So, we can have an idea about the capacities of this protection technique against errors produced in an industrial environment if we know this environment and its errors models. For example, system not far from a radioactive source corresponds to random errors whereas system near an engine that has just started induces risks of electromagnetic disturbances during this start up. This scenario corresponds to burst errors.

6. Conclusions and perspectives

In this paper, we have firstly explained the reasons of choice of stack processor architecture and the reasons of development of stack processor emulator. This emulator evaluates the internal states and the running duration and serves for architecture fine-tuning. Surely, benchmarks corresponding to processor embedded programs are developed in order to be tested in the emulator.

Because our priority is the dependability and in order to measure its impacts on time performance, we have implemented in benchmark a software protection method: when a perturbation, susceptible to plant the processor, appears, this one loops in error recovery mode and become no more functional until it returns to a stable and sure state after the occurrence of the error generator event.

This technique allows determining the border between the two types of protection (hardware and software). Some errors apparition scenarios are proposed and integrated in the emulator. The results give us ideas about the additional costs and the limits of protection of such technique for various models of injection.

As future works, we propose to test others errors models and to start developing the VHDL model.

One of the final team objectives is to evaluate and improve the stack processor dependability which will be developed and designed in order to be integrated in a multiprocessor and reconfigurable environment.

7. References

[1] H.-D. Kochs "Key Factors Key dependability of mechatronic units", 28th IEEE Annual International Computer Software and Application Conference (COMPSAC'04), September 2004.

[2] R. Isermann, R. Schwarz and S. Stolzl "Fault-tolerant drive-by-wire systems", in IEEE Control Systems Magazine, vol. 22, issue 5, pp. 64- 81, October 2002.

[3] E, Dilger, M, Gulbins, T, Ohnesorge and B, Straube "On a Redundant Diversified Steering Angle Sensor", 9th IEEE International On-Line Testing Symposium (IOLTS'03), pp. 191- 196, July 2003.

[4] E. Dilger, R. Karrelmeyer and B. Straube "Fault tolerant mechatronics", in Proc. 10th IEEE International On-Line Testing Symposium, 2004 (IOLTS'04), pp. 214- 218, July 2004.

[5] A. Shaout and T. Eldos, "On the classification of computer architecture", International Journal of Science and Technology, vol. 14, Summer 2003.

[6] Philip J. Koopman, Jr., "Stack Computers: The New Wave", California : Ed. Mountain View Press, 1989; http://www.ece.cmu.edu/~koopman/stack_computers/.

[7] J. Catsoulis, "32/64-bit zero-operand processor core for spacecraft avionics" 5th Australian space science conference, September 2005.

[8] K. Schleisiek, "scalable dual-stack processor for embedded control", in Proc. EuroForth Conf. 2004

[9] R. Mariani, P. Fuhrmann and B. Vittorelli "Fault-robust microcontrollers for automotive applications", in Proc. 12th IEEE International Symposium on On-Line Testing Symposium 2006 (IOLTS'06), pp. 213 – 218, Juillet 2006.