Bytecodes Protection Technique for Java Processor in CMOS Technology

Abstract

Real-time embedded systems have severe dependability restrictions such as: Considering the design of integrated circuits, in order to reach the reliability level required by these systems, one may use specific integrated circuits manufacturing processes or apply dependability techniques in the integrated circuit design phase. Furthermore, at software and system level, i.e. in the integration of electronic components in a printed circuit board, it is also possible to apply techniques to increase the reliability of these systems. The integrated circuits manufactured through specific processes have high cost due to their small production volume. For the same reason (for being specific), they are in general three generations late in relation to the state of the art. Therefore, it is preferable to make the design of a fault tolerant integrated circuit and be able to manufacture it using a state of the art process. From the point of view of software development, the resources provided by the programming language used to program such systems may influence the system reliability level. The use of high abstraction level languages, like Java, may diminish the occurrence of programming errors, decreasing the number of non-predicted faults inserted during the development phase. The JOP (Java Optimized Processor) soft core for FPGAs (Field Programmable Gate Array) is an optimized implementation of the Java virtual machine, in the scope of hardware, for real-time applications. In this work we propose a hardware design technique for JOP soft core, which detects and corrects errors in SRAM (Static Random Access Memory) memory code area. The occurrence of fault is detected at system level through an exception, being such feature available in Java language. The proposed technique increase system reliability and keep the features of real-time of JOP soft core.

1 INTRODUCTION

The real-time embedded electronic systems used in space missions are subject to high radiation levels existing in space. Due to that, such systems are strongly subject to faults caused by collision of heavy particles with silicon nanometric structures present in the modern integrated circuits.

Particularly for integrated circuits owning SRAM memory blocks, the main consequence of such collisions are the SEUs (single event upset), those are the permanent inversion of a bit value. Although there are foundries customized for production of radiation tolerant integrated circuits, they are highly expensive due to the small production volume of these integrated circuits. Furthermore, these foundries are, in general, two or three generations late in relation to the state of art manufacturing technology. Therefore it is very important to guarantee tolerance

to radiation in the scope of the integrated circuit design, independent of the manufacturing process, because it will reduce system costs and allow the use of the most modern existing manufacturing processes.

Due to the reduced size and high frequency of operation of the modern digital electronic circuits, they are ever more susceptible to noise. Therefore, problems formerly found only in systems submitted to radiations with amplitude similar to spatial radiation are today are present in systems operating in terrestrial environment. Security applications constitute another example of terrestrial environment that require fault tolerance techniques, because faults in security systems may be explored by crackers to find out secret keys stored in the internal memory of an integrated circuit [1].

Considering this scenario, one feels even more the need of using fault tolerance techniques not only for embedded systems in space missions, but also for terrestrial systems. Amongst some examples of these terrestrial applications we can mention automotive industry, banks and several other applications whose timing and high availability requirements are prioritized for the correct functioning of the system.

Notably, C programming language is currently the most used one for development of software for embedded systems, as for the operating system as for the application. This can be easily demonstrated by an analysis of the off-the-shelf compilers available for the modern embedded systems processors.

Usually the operating system is responsible for real-time support functions, memory management and inter-processes communication. Such resources are made available to the application by means of system calls [2] and of an API (Application Program Interface).

The use of a high level abstraction language brings benefits from the point of view of system development, such as reducing the likelihood of coding errors and the reduction of development time of a system [3]. Java is currently a high-level language largely used and with high support for standalone systems development. In addition to the high abstraction level, Java language brings on its kernel (Java Virtual Machine) resources commonly implemented in the scope of the operating system, such as inter-processes communication and tasks scheduling. In a traditional implementation of real-time embedded systems, based on a general purpose processor and a real-time operating system, these advantages have a high cost in terms of computational resources, what is incompatible to the severe constraints of computational resources in embedded systems. This incompatibility may be solved by the use of a processor customized for

Java language, as proposed by Schoeberl in [4] and others [5][6][7][8]. Nevertheless, for none of these processors the authors discuss the dependability of the processor. Compared to the remaining Java processors, JOP [4] stands out, .e.g., in relation to real-time features, but it has not incorporated fault-tolerance techniques. Due to that, we selected JOP processor as platform for this work. Accordingly, in this paper we propose a fault tolerance technique to protect SRAM code cache memory, internal to JOP, against SEUs.

2 JOP Processor

A Java processor is an implementation of the Java virtual machine. This implementation in hardware is not complete, because a Java Virtual Machine contains complex functions like, e.g., scheduling, management and process intercommunication. The cost involved in implementing all these resources in hardware can make the implementation not feasible. Therefore the concept of a Java processor differs from a general processor, in which only hardware elements are involved. Accordingly, a Java processor is an implementation based on hardware and, possibly, in some software. For a real-time Java processor, its real-time features must permeate its software as well as its hardware.

JOP (Java Optimized Processor) is a hardware and software implementation of a real-time Java virtual machine, based on profile J2ME (Java 2 Micro Edition) CLDC (Connected Limited Device Configuration) and on SCJ (Safety Critical Java) specification. This processor is implemented as soft core in Xilinx or Altera FPGAs and, differently from JVM (Java Virtual Machine), that is a CISC (Complex Instruction Set Computer)[9] machine, JOP is internally a RISC (Reduced Instruction Set Computer) [9] machine, and therefore contains its own instructions set.

2.1 Implementation of JVM in JOP

On JOP, Java bytecodes are decoded through a pipeline of equivalent native JOP instructions. For some Java bytecodes, there is a bi-univocal equivalence to JOP native instructions, which run during a single cycle of clock. Medium complexity bytecodes are translated to a sequence of JOP native instructions, found in a table contained in a ROM (Read Only Memory) memory area, called JVM microcode or Java Virtual Machine Microcode. More complex bytecodes, like e.g., the instruction **new**, are implemented on the own Java language and, therefore, translated to sequences of the remaining bytecodes, in running time. In order to optimize the performance of customized instructions, it is possible to implement them in hardware. JOP, similar to original

JVM, is a stack machine, i.e., instead of doing operations over a set of registers, as it occurs in x86 architectures, the operations are done over the items on the top of the stack.

2.2 Interrupts and exceptions

Interrupts are used for signaling external events, e.g., to detect that a button was pressed. When an interrupt occurs, the processor simply stops running the code currently pointed by the program counter register, and deviates the execution to an interrupt routine. Furthermore, the context of the interrupted process is stored in order to be restored later. It includes storing CPU registers and processor status registers. These actions make possible the return to the execution of the original code when the interrupt routine was ended.

In JOP, the interrupts and exceptions generate special bytecodes (sys_int and sys_exc), which are inserted by the hardware transparently, in the sequence of bytecodes to be executed. Interrupt handlers may be implemented in a similar way to bytecodes, i.e., in microcode or Java [10]. Below follows an example of code depicting how the programmer should implement an interrupt handling.

```
Public class CACHEReceiveInterruptHandler
extends InterruptHandler {

    private CRCCACHE crccache;
    private InterruptCtl interruptCtrl;

    public CRCCACHEInterruptHandler
    (CRCCACHE crcache,InterruptCtrl interruptCtrl)
    {
        // Registra manipulador
        // de Interrupção
        super(INT_CRCCACHE);
        this.crccache = crccache;
        this.interruptCtrl = interruptCtrl;
    }
```

```
    protected void handleInterrupt() {
        // Levar o sistema a um
        // modo de falha seguro
    }
}
```

2.3 Real-time requirements

Real-time applications for JOP are explicitly split in two parts: Initialization Phase and Mission Phase. In the initialization phase every object to be used during all the execution of the application is created and, therefore, memory areas are allocated and initialized. In this phase there is no real-time warranty. In mission phase, the threads are executed concurrently according to the scheduling algorithm.

2.3.1 WCET Analysis in JOP

Due to JOP has been developed to be used in embedded systems with real-time applications, this processor architecture allows to calculate easily the WCET (Worst Case Execution Time) of a task.

JOP Java virtual machine implements classes which allow developing real-time applications. These classes are not compatible with RTSJ (Real Time Specification for Java) [11] standard, because only a subset of this standard is implemented. Although code and stack areas of JOP use cache memory, the modeling of cache memory behavior in JOP is perfectly predictable in time. This is due to the fact that, differently from other processors, “cache misses” in JOP cache do not occur, i.e., each instruction fetched by the processor in cache will necessarily be previously stored in cache.

2.3.2 Dependability

JOP does not implement techniques of dependability in its architecture. Therefore, for hard real-time applications, i.e., those which involve risk for human lives, the system designer shall assure the dependability at system level. On JOP, a hardware fault, e.g., an illegal opcode or a memory parity error, will take the system to a shutdown [12].

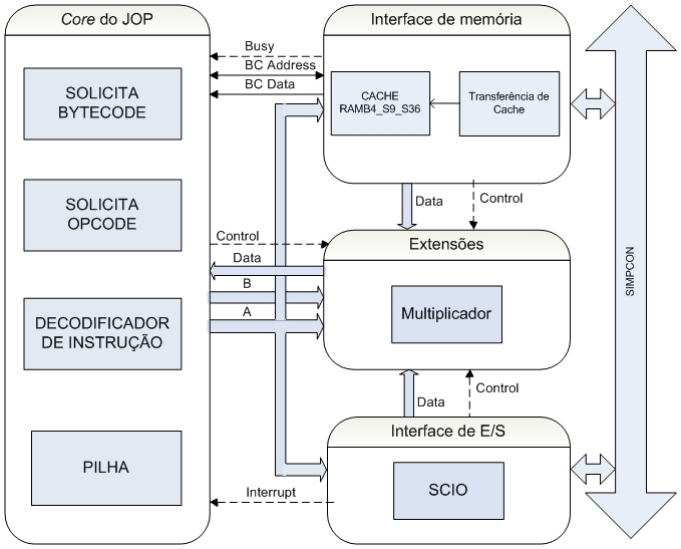

Fig. 1. JOP Block Diagram

2.4 JOP Architecture

JOP is composed of four main blocks (see Figure 1): Memory interface, JOP core, I/O interface (**scio**) and extensions. The memory interface block communicates with the controllers of external memory through the communication bus **simpcon**. The memory controllers, in turn, communicate through the processor pins with SRAM and flash memories. The JOP core block is responsible for decoding and running the bytecodes provided by the memory interface and for commanding the remaining parts of the processor. The I/O interface block communicates with the I/O controllers, such as USB port and RS232 serial interface, through **simpcon** bus. These I/O controllers, in turn, communicate with the external environment devices through the processor pins. The "extensions" block serves to aggregate functions of mathematical co-processors without doing modifications on the processor kernel.

2.5 Documentation and portability

In addition to the JOP technical features described earlier, we may also highlight its comprehensive available documentation, its portability, to the extent that it was already implemented in several FPGA development boards (Xilinx and Altera) available in the market, and last, but not least, the availability of JOP source code to download through site http://www.opencores.org and licensed under GPL (Gnu Public License) version 3. Currently JOP is used in two off-

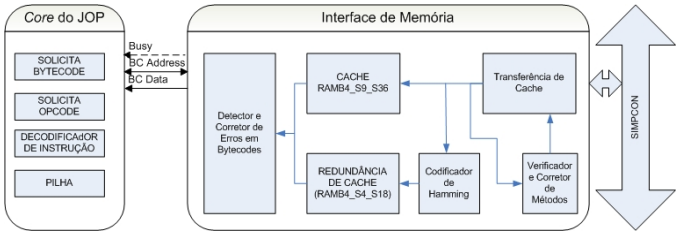

Fig. 2. Block Diagram of modified JOP

the-shelf systems, one of them having the real-time feature, and in several research systems [4]. Therefore JOP positions itself as an excellent option of research and development basis platform for new techniques of real-time systems.

3 FAULT TOLERANT JOP

Errors on code memory of a computational system are critical due to being stored permanently and may, therefore, cause successive errors in the computing process that makes use of wrong data. To detect and correct errors in data memory, there are very effective techniques implemented in software. Nevertheless, the techniques applied in the scope of software to detect and correct errors in code memory are not effective, because the software itself may be corrupted. In this sense, we propose the use of a technique in the scope of hardware to detect and correct errors in the RAM (cache) memories internal to the JOP processor, as a way of increasing the reliability of the system, particularly the cache of methods described in the previous section.

During the initialization of JOP processor, every code is transferred from flash to RAM memory. At the end of code transfer to RAM, the cache system will transfer, on demand, entire methods from external into internal cache memory. Lastly, when the methods are completely stored in internal memory, the core will ask and execute the bytecodes, one by one.

The Figure 2 depicts the modified JOP block diagram. When compared to the original JOP block diagram (see Figure 1), one notices that three new blocks were added to the block of memory interface of the JOP original architecture. These blocks refer to the implementation of the instructions protection technique:

1) Hamming Coder;
2) Error detector and corrector in bytecodes (Hamming Decoder);
3) Cache redundancy (RAMB4_S4_S18);

3.1 Technique of instructions protection

The technique described above detects and corrects errors occurred on bytecodes, from the moment they are stored into the cache until the beginning of their execution by JOP core. In fact, the errors are detected and corrected immediately before the core begins bytecode execution. Therefore, this is a *last-moment* check [13] .

Simultaneously to the writing of a bytecode (sized 8 bits) on cache memory, four redundancy bits (Hamming bits) are calculated and stored in a redundancy cache memory (see Figure 2). These extra bits are calculated by a core written in RTL (Register Transfer Level) which implements a Hamming coder.

Immediately after cache memory provides a bytecode to JOP core, but before being executed, a Hamming decoder core reads the 4 bits stored on redundancy cache (see Figure 2). Based on these 12 (twelve) bits (eight bits of bytecode plus four bits of Hamming), this core will check if there were any bit inversion in any of the bytecode bits. In affirmative case, this means there was a fault. In this case, the Hamming decoder core will correct automatically the bytecode, since only one bit has been inverted. Finally, the correct bytecode will be delivered to the execution by JOP core.

3.2 Perception of fault in the system scope

A hardware fault in the original JOP architecture, like e.g., an illegal opcode, takes the system to a shutdown [12]. In this work, the processor was modified in order to, on the occurrence of a hardware fault, an exception be generated. Below follows a code example of how the programmer should build the treatment of a hardware fault.

```
static int saved_sp;
// Executado em caso de uma exceção
// gerada pelo hardware
static void except() {
    saved_sp = Native.getSP();
    if (Native.rdMem(Const.IO_EXCPT)
    ==Const.EXC_CRCCACHE)
    {
```

TABLE 1

Comparative Table between Modified JOP and Original JOP

	FPGA Slices	RAM (Kbits)	Freq (MHz)
JOP	3150	16	100
JOP e Hamming	3201	24	100

```
        // Tratamento da exceção
        handleException();
    }
}
```

4 RESULTS

The JOP modified by the use of the technique of instructions protection was simulated using the NC-VHDL tool to evaluate if there were depreciation on its functioning. To evaluate the efficiency of the technique, a fault injector module written in VHDL was developed. This module generates random events of SEU (no more than one bit inverted on the same bytecode) type on the bytecodes of a program in execution by JOP. The technique proved itself efficient, since that every corrupted bytecode with only one inverted bit was automatically corrected. To those ones with two inverted bits, an exception was generated to be treated by a routine aimed to this purpose. Besides simulation, the modified JOP was embedded on Spartan-3 starter kit board [14] from Digilent. This board contains a 200,000-gate Xilinx Spartan-3 XC3S200 FPGA (XC3S200FT256) and 1MB SRAM memory. In that case, the fault injection was done through changes of the logical level of FPGA pins. Two kinds of faults were injected: stuck-at (1 or 0) and inversion of random bits. For every inserted fault, whichever kind, there were detection and automatic correction due to the use of the instructions protection technique.

The Table 1 compares the area of FPGA, in terms of slices and RAM memory for the original JOP and the JOP modified by the proposed technique.

5 CONCLUSION

The proposed technique aggregates to the JOP processor the capacity of detecting and correcting errors in code memory. This fact allows the use of the processor together with ordinary memories

like SRAM without ECC (Error Correcting Code), that are manufactured in large scale, and therefore have a lower cost per silicon area.

It is interesting to note that the proposed technique may be applied to Java processors implemented in FPGA or in ASIC. In the event of a multiprocessing system based on JOP and implemented in FPGAs, this technique may be combined with the one proposed by Castro et Al in [15].

REFERENCES

[1] C. R. Moratelli, Érika Cota, and M. S. Lubaszewski, "A cryptography core tolerant to dfa fault attacks," *Journal Integrated Circuits and Systems*, vol. 2, no. 1, pp. 14–21, 2007. [Online]. Available: http://www.sbmicro.org.br/jics/html/artigos/vol2no1/02-Moratelli-v2n1.pdf

[2] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Pearson Education, 2008.

[3] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1991.

[4] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. doi:10.1016/j.sysarc.2007.06.001, 2008. [Online]. Available: http://www.jopdesign.com/doc/rtarch.pdf

[5] A. C. Beck and L. Carro, "Low power java processor for embedded applications," in *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

[6] T. R. Halfhill, "Imsys hedges bets on Java," *Microprocessor Report*, August 2000.

[7] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer, "Real-time event-handling and scheduling on a multithreaded Java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19–31, 2003.

[8] W. Puffitsch and M. Schoeberl, "picoJava-II in an FPGA," in *Proceedings of the 5th 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. Vienna, Austria: ACM Press, September 2007, pp. 213–221. [Online]. Available: http://www.jopdesign.com/doc/pjfpga.pdf

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed. Morgan Kaufmann, 2007.

[10] S. Korsholm, M. Schoeberl, and A. P. Ravn, "Java interrupt handling," in *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. Orlando, Florida, USA: IEEE Computer Society, May 2008. [Online]. Available: http://www.jopdesign.com/doc/ihjava_isorc2008.pdf

[11] P. Mikhalenko, "Real-time java: An introduction," ONJava.com, September 2008. [Online]. Available: http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html?page=1

[12] M. Schoeberl, *Jop Reference Handbook*, 1st ed., 2007.

[13] H. de Sousa Castro, "Fault tolerance through reconfigurability: Applications in space instrumentation," Phd Thesis, The University of Sussex, June 1992.

[14] *Spartan-3 Starter Kit Board User Guide*, Xilinx, Inc., May 2005, v1.2.

[15] H. Castro, A. A. Coelho, and R. J. Silveira, "Fault-tolerance in fpga's through crc voting," in *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits And System Design*. New York, NY, USA: ACM, 2008, pp. 188–192.