

# A Fault-Tolerant Approach to Embedded-System Design Using Software Standby Sparing

Mehdi Modarressi, Hani Javanhemmat, Seyyed Ghasem Miremadi, Shaahin Hessabi, Morteza Najafvand, Maziar Goudarzi, and Naser Mohamadzadeh  
Computer Engineering Department,  
Sharif University of Technology, Tehran, Iran.

{modarressi, javan}@mehr.sharif.edu, {miremadi, hessabi, goudarzi}@sharif.edu, {najafvand, naser}@mehr.sharif.edu

## ABSTRACT

A fault-tolerant approach for application-specific instruction-set processor (ASIP) to reduce the cost of classical fault-tolerant mechanisms is presented in this paper. The ASIP is synthesized from an object-oriented high-level description by a hardware-software co-design approach and consists of a processor core along with some functional units. In the proposed fault-tolerant methodology, upon detecting a fault in a hardware functional unit, this unit is replaced by an equivalent software version. To evaluate the fault-tolerant approach, a JPEG processor is used. The analytical and experimental results show that the approach can tolerate functional-unit faults with no area/cost overhead. However, by using the functional-unit fault-tolerant approach along with the traditional duplication approach, we can make a tradeoff between area overhead and performance degradation. In addition, the processor core of the ASIP has been made fault-tolerant with lower area overhead than the traditional duplication method by putting a simpler application specific core as a redundant module for the main processor. Keeping only the data paths used by the running application in the redundant core enables us to reduce the redundancy cost with no performance overhead.

## Keywords

Embedded systems, Fault-tolerant systems, Standby sparing.

## 1. Introduction

Embedded systems are spreading out to various areas of our everyday life. Such systems are all around us in our cars, homes, workplaces, and even in our pockets. Some of them are simply accessories or entertainment equipments (e.g. MP3 player or PDA) while some others function in safety critical applications such as Anti-Lock Braking Systems (ABS). No matter what an embedded system does, it is increasingly important to make it fault tolerant. While for safety critical applications this is a must, it is desired even in non-critical applications so as to ensure satisfactory operation or reasonable quality of service.

*Redundancy* is the fundamental mechanism to tolerate faults in a general system [1]; redundant hardware modules are put as hot or cold spares and a voting or checking mechanism is used to detect and tolerate faults. This general solution works fine, but inherently imposes additional costs and/or area/power overheads due to replication of modules. The same general mechanism can be used to tolerate faults in those embedded systems that comprise a combination of hardware and software components. Such hardware-software embedded systems are becoming more popular today since they allow the designer to take advantage of the synergy between hardware and software implementation styles to meet today sophisticated design criteria such as performance, power consumption, area and cost.

In this paper we first introduce our embedded system architecture that we follow in this research. The embedded system is synthesized from a high level object-oriented specification. The whole system comprises one or more processor cores in addition to some hardware functional units. The functional units are synthesized from some of the class methods in the object-oriented input specification. The other class methods are implemented in software as software routines [2]. We first present an approach to tolerate the functional-units faults with negligible cost overhead. Reducing fault tolerant cost overhead at the functional unit level, has been applied mostly in FPGA-based reconfigurable systems [3], [4], [5]. These methods often use one or more spare units and configure it to perform the faulty functional unit tasks when detecting a fault in a functional unit. However, [6] proposes a fault-tolerant method using functional unit redundancy in non-FPGA general purpose processors by putting a redundant functional unit for a group of similar functional units.

However, the presence of software as well as hardware in our systems enables new approaches to tolerating faults. An approach that we pursue in this research uses software routines to replace faulty hardware units. These software replacements are executed on the same processor that is already available in the system, and hence, no area and cost overhead is imposed; however, the overall time for the system to finish its designated task is obviously affected. In other words, instead of paying extra costs for redundant hardware units, that affects the system cost irrespective of fault rate in the working environment of the system, we achieve fault-tolerance by paying extra time for the system task execution which, conveniently, only is incurred in the special case where a permanent fault actually happens.

Obviously, this fault-tolerant method reduces the processor performance. We will analyze this performance degradation and show that by using the proposed method along with the hardware standby sparing, we can minimize the hardware cost for any given target performance.

In addition to functional unit fault-tolerant design, we present a method for reducing the cost overhead of the processor core fault-tolerance. To do so, we use an application-specific processor core as redundant module for the main processor core. This application-specific processor core is made by removing the unused data paths from the processor soft core. [7] applies a similar method for fault-tolerant design. However, [7] makes new data paths by combining the used instructions in the running application. Although having low area overhead, it only can tolerate data path (and not the control) faults. Designing application-specific soft cores have also been used in designing embedded systems soft cores [8], [9].

In the rest of this paper, Section 2 reviews related approaches to fault tolerant system design. In Section 3, the embedded systems that we use in this research and fault detection and tolerant

mechanisms are introduced. In Section 4, we mathematically analyze the proposed fault-tolerant approaches. Section 5 evaluates the fault-tolerant approaches by a case study. Finally, Section 6 summarizes and concludes the paper.

## 2. Related work

Several hardware and software methods for tolerating faults in computer systems have been developed and utilized in commercial processors.

The main idea of the most of the methods is based on the physical replication of hardware. There are three basic forms of hardware redundancy: *passive*, *active* and *hybrid techniques* [1]. Passive techniques mask the fault and prevent it from resulting in errors. An example of this form is the well-known Triple Modular Redundancy (TMR) technique which triplicates the hardware and performs majority voting to determine the output.

Active techniques do not prevent the fault from producing errors within the system, but detect and locate the fault for recovery. These techniques are suitable for systems that can tolerate temporary wrong results as long as the system reconfigures and regains its operational status. *Hot-standby* and *cold-standby* [1] are the most famous examples in this category. Finally, the hybrid approach applies both active and passive methods to tolerate faults.

Since our approach is based on replacing faulty functional units by software equivalents, in this section we review some fault-tolerant mechanisms that work based on functional unit redundancy.

A method for designing fault-tolerant processors using functional unit redundancy is proposed in [6]. The proposed method arranges functionally similar blocks in groups. For each of these groups, a backup super block is defined which covers the function of all group members. This method can reduce the area overhead of fault-tolerance. However, since the hardware is optimized in processors to avoid unnecessary overhead, it is difficult to find similar functionality among functional units.

In addition to the above method, some fault-tolerant approaches are proposed for FPGA-based systems that implement functional block redundancy by run-time reconfiguration.

A fault-tolerant embedded system design methodology that uses dynamically reconfigurable FPGAs as spares for several dedicated hardware components in real-time applications is presented in [3]. The advantage is the reduction of area or cost as compared to dedicated spares. During normal operation, each FPGA is dynamically reconfigured with system tasks and can replace any of them if a fault is detected. For a specified coverage, i.e., system tasks that might be affected by a single fault, the algorithm allocates a set of FPGAs and determines schedules, including task executions and FPGA reconfigurations that provide the required redundancy while satisfying deadlines and minimizing either area or cost.

In [4], a computer aided design approach to synthesize Application Specific Programmable Processors (ASPP) is presented that can tolerate single and multiple functional unit failures. In this method, fault-tolerance can be incorporated at the behavioral level by combining the flexibility of multiple applications in an ASPP with judicious application-to-faulty-unit assignment. Using the architectural flexibility of an ASPP, this method develops approaches for graceful degradation-based permanent fault-tolerance. Graceful degradation is supported by

implementing multiple schedules of the ASPP applications, each with a different throughput constraint.

In this paper we present a new method for functional block redundancy by using software as spares for hardware functional units. To the best of our knowledge, this is the first technique that replaces faulty hardware by software. As we will discuss in the following section, applying polymorphism in hardware enables us to use this mechanism, while it is hard (or impossible) to implement this mechanism in systems without this ability.

We also use an application-specific processor core as a redundant module of the ASIP processor core. An approach which applies application-specific processor core for fault-tolerance is proposed in [7]. Owing to the fact that the embedded software often remains unchanged during system life time and uses only a subset of the processor instruction set, this approach analyzes the running software of the system and puts a coprocessor containing the used instructions in the application. The area of the coprocessor is reduced by finding the similar micro-operations in the selected instructions and implementing only one instance of each micro-operation. Some other papers concentrate on deriving application specific processors from soft processor cores in order to reduce the complexity and overhead of the embedded systems (but not to tolerating faults). For example, [8] removes unused functionalities from a soft processor core by adding an extra step, core optimization step, to the system design steps. Similarly, [9] introduces some approaches such as soft processor cores, configurable processors, and ISA extensions, for architecture optimization of embedded systems.

In this paper we develop an application specific processor core by removing unused datapath elements from a soft processor core. We use this core as a redundant module for the processor core of our embedded system.

In the next section, we introduce the embedded processor architecture which we use in this research and the details of the proposed fault-tolerant approach.

## 3. Fault-tolerant approach

### 3.1 . Embedded system architecture

The embedded system architecture that we follow in this research is depicted in Fig.1. The system is a Network-on-Chip (NoC) architecture that consists of a processor core along with a set of hardware functional units (FU) [2].

The architecture is specifically designed to suit object-oriented (OO) applications. A typical OO application defines a library of classes, instantiates objects of those classes, and invokes methods of those objects. Our implementation approach for each of these three major components of an OO application is described below. For presentational purposes, we follow the C++ syntax in describing each component.

**Class library:** Each class consists of variable declarations and method definitions. Variable declarations are compile-time information and do not require a corresponding in the implementation. Methods of the OO class library are either implemented in software (e.g.  $A::g()$  and  $C::f()$ ) in the "Instruction Memory" box in Fig.1) or in hardware (e.g.  $A::f()$  and  $B::f()$ ) FUs below the "Processor core" box in Fig.1).

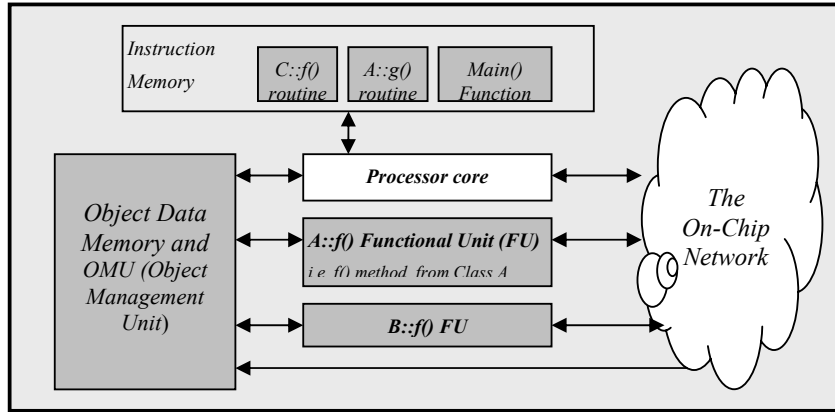


Fig. 1. The internal architecture of our embedded systems

in the implementation. Methods of the OO class library are either implemented in software (e.g.  $A::g()$  and  $C::f()$ ) in the “Instruction Memory” box in Fig.1) or in hardware (e.g.  $A::f()$  and  $B::f()$ FUs below the “Processor core” box in Fig.1).

**Object instantiations:** these are specified in the  $main()$  function. A memory portion should be reserved for each instantiated object to store the values of its data items. This memory portion is allocated in a data memory (the gray box at the left-hand side of Fig.1) that is accessible to the processor core as well as all FUs.

**Method invocations:** the sequence of method invocations is specified in the  $main()$  function of the application. The executable code of this function comprises another part of the instruction memory (see Fig.1).

The processor core starts by reading the instructions specified in the  $main()$  function of the application. Whenever a method call instruction is read, the corresponding implementation is resolved and invoked. This may result in calling a software routine (e.g.  $C::f()$  in Fig.1) or activating an FU (e.g.  $A::f()$  in Fig.1). Each method implementation (be it in hardware or software) can also call other methods. Since methods may be implemented in hardware as well as in software, new techniques are required to efficiently dispatch method-calls among method implementations. To achieve this goal, we view each method call as a network packet. Each method call is identified by a called method, a called object and the parameters of the call. Therefore, each packet contains fields named  $mid$ ,  $oid$  and  $params$  to respectively represent called method number, caller object number and call parameters. The numbers allocated to methods and objects are assigned such that routing the packet on the on-chip network corresponds to dispatching the call to the appropriate called method irrespective of the caller and called being in software or hardware [10].

To implement polymorphism, we simply set the value of the  $oid$  part of the packet at run-time and send it on the network; depending on the dynamic  $oid$  value, the packet may reach different FUs, but in all cases it will be the appropriate one due to the FU address assignment scheme [2].

The details on resolving method calls and polymorphism, passing parameters, synchronizing hardware and software, and other details of the architecture can be found in [2] and [10].

### 3.2 . Processor core fault-tolerant mechanism

As mentioned earlier, our embedded system consists of three parts: a processor core, some functional units and an on-chip

network. We have already proposed a mechanism to detect and tolerate the on-chip network faults [11]. In this part of the paper, we present a fault-tolerant mechanism for the processor core.

We use an existing embedded processor as the processor core of the system. Although any existing processor such as PowerPC or ARM can be used, we have selected the LEON processor [12] as the processor core of our embedded system . LEON is a 32-bit processor conforming to the SUN SPARC architecture. Fig.2. shows the block diagram of a subset of the LEON processor core used in this research.

The main focus of this section is to present a fault-tolerant mechanism for the integer unit of this processor core. The integer unit in the LEON processor is implemented as a 5-stage pipeline. A floating-point co-processor along with the integer unit enables the processor to perform floating point operations.

We use LEON soft processor core to develop an application specific redundant module for the main processor core of the system. Soft cores enable the designer to customize the processor core by selecting the instruction set, adding new instructions, and changing the cache and register file sizes [13] [14].

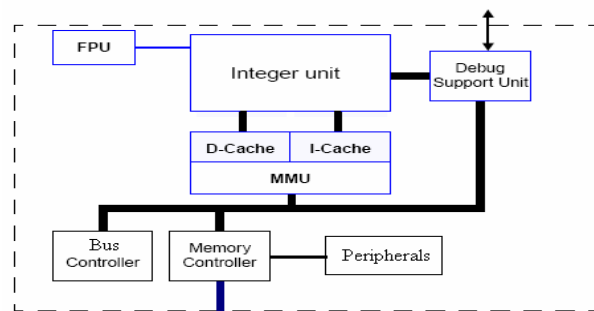


Fig. 2. A subset of the LEON processor core which we used in this research

As mentioned earlier, an embedded application is specific and remains unchanged during system life time. Owing to this fact, we can extract the LEON instructions that are used in the machine code of the running application. Then, we can make another simpler core by removing the unused instructions from the soft core of the processor. Afterwards, we will synthesize and insert this simpler core in the system as an application-specific redundant module for the main processor core. The processor is replaced by the redundant module after detecting a

permanent fault and the input and outputs are diverted to the new core by a simple multiplexing circuit.

As long as the application corresponding to the redundant module is unchanged, this method guarantees that the system continues operation in presence of a detected permanent fault in the processor core. Our method may much better fit regular architectures such as LEON than highly irregular and complex processors. The regular structure of LEON processor core allowed us to easily detect and eliminate unused datapath elements from the core. More aggressive elimination can be applied by removing all other parts (e.g. parts inside the control unit) that correspond to these datapath elements, but we only followed the former approach since datapath elements are expected to consume more area than their control unit counterparts.

In addition to the integer and floating-point units the processor comprises two cache memories and one external memory management unit, a debugging circuit, and some peripherals. This paper does not concentrate on the fault-tolerance of these components. However, any conventional fault-tolerant method can be used to tolerate their faults. For example, [15] and [16] propose two mechanisms to tolerate the cache memory faults. Applying the method proposed in [15] imposes only 10% area overhead to the system.

In the following sections, we will evaluate this processor core fault-tolerant approach by a case study.

### 3.3 Hardware functional units fault-tolerant mechanism

#### 3.3.1. Fault detection mechanism

We take advantage of *assertion checkers* for online detection of faults in the system. The assertions are specified by the designer of the embedded application and are synthesized to either *assertion checker circuits* for assertions on functional units, or *assertion checker codes* for assertions on software routines. This assertion synthesis process is accomplished by our design automation tool that is currently being developed.

The assertions that we currently envision, fall into two categories. *Combinational assertions* can be arithmetic, logical, or comparative statements involving any combination of variables, constants, and return value of method-calls. *Sequential assertions* specify statements on the expected sequence and conditions of method-calls. We rely on the intuition of the application designer in specifying enough and correct set of assertions to ensure correct operation of the final embedded system. Since the set of detectable faults in this approach covers those faults that raise an assertion failure, the designer determines target faults by adding and/or removing assertions in the application. Further details on the type of assertions, and the approach to synthesizing them can be found in [17].

In order to distinguish permanent from transient faults, if the output of a functional unit violates the assertion, the functional unit will run again. If the problem persists we assume that the fault is permanent and initialize the replacement process. If the second execution does not violate the assertion we repeat the execution for the third time. If the third results agree the second results we conclude that the fault was transient and do not

replace the functional unit; otherwise, we assume that the fault is permanent and initialize the replacement process which is described in the following section.

#### 3.3.2. Fault detection mechanism

In this section, we propose a mechanism to tolerate functional unit faults. To tolerate functional unit faults, we take specific advantage of OO polymorphism and the way that it is provided in our embedded system architecture. Polymorphism implies that when a method is called on an object that can dynamically change its class (i.e., a so-called *polymorphic object*), the invoked method implementation depends on the run-time class of the object. For example, assuming that class *B* is derived from class *A* and a polymorphic object, say *ap*, is defined that can dynamically change type to either *A* or *B*, the *ap->f()* method-call is either dispatched to *A::f()* or *B::f()* depending on the class of *ap* when the *ap->f()* is executed. Since our embedded system architecture allows each method implementation to reside either in hardware or in software (see Fig.1 and Section 3.1), we can choose to implement *B::f()* in software with exactly the same functionality as *A::f()* (which is implemented in hardware). Thus, if *A::f()* is found to be faulty (see Section 3.4.1), it can be replaced with its software equivalence (i.e. *B::f()* method) by simply changing the type of objects at run-time. The same can be done for all hardware units that are desired to tolerate faults.

For example, Assume, as above, that class *B* is derived from *A* and both of them have defined the *f()* method while *B::f()* runs in software the same function that *A::f()* runs in hardware. Further assume that a polymorphic object *ap* is defined that can dynamically change type between *A* and *B*. The following pseudo-code demonstrates our fault-tolerant approach of replacing faulty hardware units by their software equivalents:

```
// class of ap is A
ap->f();
if (an assertion failed)
{
    change class of ap to B;
    ap->f(); // redo f() in sw
}
```

Since the assertions are checked at run-time, the above *if* statement actually provides the desired fault-tolerance. The statement that changes the class of a polymorphic object simply changes the hidden tag that identifies the class of *ap*; this hidden tag is stored in the data memory of Fig.1 in the same portion that is allocated for other data of the *ap* object. Note that the re-executing procedure explained at the final paragraph of Section 3.4.1 is not reflected in the above code excerpt.

The above-explained fault-tolerant scheme obviously puts some performance overhead to the system. In the following section of this paper we analyze this overhead.

### 4. Mathematical analysis

This section focuses on analyzing the fault-tolerant methodology for the functional units and comparing it with other classical approaches, i.e., TMR and Standby-Sparing. For this reason, we first develop a number of formulas to model the methodology

and estimate its performance and cost overhead. Then, we evaluate them by an example.

As mentioned earlier, running program (main function) consists of a number of method invocations which are implemented either in hardware or software. As a result, we can model the system as a list of method invocations.

Let  $N_i$  denote the number of invocations of method  $M_i$  in an interval of time. This interval can be the time needed to perform a task, such as the time needed to process an input picture in a JPEG processor or a reasonable time span in systems that continuously process an input stream, such as an engine control unit in a car. We also define  $T_{H,i}$  as the average execution time of method  $M_i$  implemented as a hardware functional unit and  $T_{S,i}$  as the average execution time of method  $M_i$  as a software routine running on a processor. In addition, we define  $S_{hw}$  as the set of hardware implemented methods and  $S_{sw}$  as the set of software implemented methods.

The execution time of the program in the period mentioned above, for fault-free system (without any redundancy) is:

$$T_{total} = T_{main()} + \sum_{i \in S_{sw}} N_i \times T_{S,i} + \sum_{i \in S_{hw}} N_i \times T_{H,i}$$

Where,  $T_{main()}$  is the execution time of the  $main()$  function which includes the time of object instantiation, controlling the sequence of method calls, input/output operations and so on.

The execution time for standby sparing is also as above. However, for the TMR we have the following relation, since a voting time will be added to the execution time of the hardware functional units:

$$T_{TMR} = T_{main()} + \sum_{i \in S_{sw}} N_i \times T_{S,i} + \sum_{i \in S_{hw}} N_i \times (T_{H,i} + T_v)$$

Where,  $T_v$  is the voting time among three functional units.

The total execution time of our proposed mechanism ( $T_{sw \rightarrow hw}$ ) after detecting a fault and replacing the faulty functional unit is:

$$T_{sw \rightarrow hw} = T_{main()} + \sum_{i \in S_{sw}} (N_i \times T_{S,i}) + \sum_{i \in S_{hw}, i \neq f} (N_i \times T_{H,i}) + N_f \times T_{S,f}$$

Where,  $M_f$  is the faulty functional unit.

If we define  $\Delta T_i = T_{S,i} - T_{H,i}$ , the performance degradation ( $PD$ ) of the proposed mechanism is:

$$PD = \frac{T_{total}}{T_{sw \rightarrow hw}} = \frac{1}{1 + \left( N_f \times \frac{\Delta T_f}{T_{total}} \right)}$$

Regarding the above formulas, any increase in the number of invocations or the execution time of the faulty functional unit(s), increases the performance overhead of the faulty system.

On the other hand, the hardware cost of our fault-tolerant mechanism ( $C_{sw \rightarrow hw}$ ) is:

$$C_{sw \rightarrow hw} = C_{processor} + \sum_{i \in S_{hw}} C_i$$

Where,  $C_{processor}$  is the hardware cost of processor and  $C_i$  is the hardware cost of the functional unit corresponding to method  $M_i$ . Since the software versions of the functional units are stored in the instruction memory, our mechanism has no hardware overhead for the system. Consequently, this hardware cost is identical to a simple system which has no redundancy ( $C_{no-redundancy}$ ). Since we use the instruction memory to store the software equivalent of functional units, no additional hardware is needed. However, it may be needed to use a larger memory module. We will analyze this memory usage later.

For the TMR, the hardware cost is:

$$C_{TMR} = C_{processor} + \sum_{i \in S_{hw}} (3 \times C_i + C_{voter})$$

Similarly, the hardware cost for standby sparing is:

$$C_{Standby} = C_{processor} + \sum_{i \in S_{hw}} (2 \times C_i)$$

While our method has no hardware overhead for the system, the hardware for the TMR is

$$\left( 1 + \frac{\sum_{i \in S_{hw}} ((2 \times C_i) + C_{voter})}{C_{no-redundancy}} \right) \text{ times larger than}$$

the hardware of a system having no redundancy.

Likewise, the hardware cost of standby sparing is

$$\left( 1 + \frac{\sum_{i \in S_{hw}} C_i}{C_{no-redundancy}} \right) \text{ times larger than the hardware of}$$

a system having no redundancy. Obviously, our fault-tolerant mechanism reduces the hardware cost of fault-tolerance.

#### 4.1 Processor core

In order to make the processor core fault-tolerant, we can apply some traditional approaches such as standby sparing or the TMR. Using standby sparing, the following overhead will be imposed to the system:

$$C_{Standby-sparing} = 2 \times C_{processor} + \sum_{i \in S_{hw}} (C_i)$$

Likewise, Using the TMR, results in the following overhead:

$$C_{TMR} = 3 \times C_{processor} + \sum_{i \in Shv} (C_i)$$

However, since the area overhead of the fault-tolerant approach presented earlier in this paper (see Section 3.2) depends on the application running on the processor and the processor core structure, we will discuss it in the following section using a case study. Since we remove only unused instructions in the redundant module, the proposed fault-tolerant approach has no performance overhead for the system.

## 5. Experimental results

In this section, we evaluate the above formulas by a real ASIP. As an example, we have selected a JPEG processor. This processor is synthesized by “Synopsys SystemC Compiler” using class\_fpga.db library [18]. The LEON application specific core is also synthesized by Synopsys using the same library. The clock rate of the processor cores and the functional units are the same.

First, we assess the functional-units fault-tolerant approach. Table.1 shows the number of invocations, the execution time, and the area of the JPEG class methods which are implemented as hardware functional units. The “number of invocations” field in the table indicates the number of invocations of a method in processing a 320×240 input bitmap picture. In this ASIP, there are 15 functional units along with the processor core. As mentioned earlier, we evaluate the performance overhead of our fault tolerant mechanism in processing a 320×240 input picture which takes 16,888,452 clock pulses to be executed completely. The size of the software versions of the hardware functional units is 5100 bytes. We can store them in the current instruction memory and do not need any additional memory module. Fig.3 and Table 2 illustrate the performance degradation of the JPEG processor ( $T_{\text{fault-free}}/T_{\text{faulty}}$ ) when replacing one of the functional units because of a fault.

The results show that replacing most of the functional units by software versions has a low overhead for the processor performance. However, as we expect (regarding the mathematical analysis), a fault in a large functional unit leads to more performance overhead. Similarly, the performance overhead increases with the number of the faulty method invocations. In this example, replacing the methods  $M_1$ ,  $M_6$ ,  $M_7$ , and  $M_{13}$  reduces the performance by more than 35%. If this overhead is not tolerable by the system, we can use a hardware standby spare for these methods, while keeping software spares for the other functional units. In this case, 65771 gates are imposed to the system as redundant functional units. This hardware overhead is 185745 gates for the standby sparing and 371490 gates for the TMR. As a result, our hardware overhead is

35% of the overhead imposed by the standby sparing and 17% of the overhead imposed by the TMR.

**Table1. Method characteristics of the JPEG processor**

Method id	Number of invocations	Execution time (clock pulse)	Equivalent Software Execution time	Area (number of gates)
0	3720	320	2260	4318
1	3720	528	7065	17378
2	1	42	880	13059
3	1	2752	11007	21113
4	1	1780	19369	10230
5	1	11223	104445	14407
6	238080	16	740	3367
7	1	998404	14382156	14909
8	1240	848	7920	17655
9	1240	16	647	5049
10	1240	448	3819	10371
11	1240	448	3819	10371
12	1240	448	3819	10371
13	3720	472	5952	30117
14	1	21	1005	3029

This is a trade off between performance degradation and hardware cost of redundancy. On one hand, by using hardware spares for large functional blocks we can reduce the performance overhead; but on the other hand, it requires more redundant hardware. By selecting an optimum point (regarding the performance overhead that the system can tolerate) we can save most of the hardware required for redundancy, while we are sure that the performance meets any given minimum degree, if a fault occurs.

In addition to the above benefits, our fault-tolerant approach can be used to tolerate design faults. If there is a problem in a functional unit design, the TMR and standby sparing can not detect and tolerate the fault. However, by our method, we can prepare a correct software version of the functional unit and redirect the faulty method invocations to the counterpart software version. Similarly, this methodology allows the designer (even user) to update the functionality of the hardware functional units by new software versions.

After assessing the functional-units fault-tolerant approach, we evaluate the proposed fault-tolerant method for the processor core. The LEON processor has 73 instructions. Analysis results

**Table2. Degradation ( $T_{\text{fault-free}}/T_{\text{faulty}}$ ) when a method is replaced by its software equivalent.**

Method id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Performance degradation	0.70	0.41	0.99	0.98	0.99	0.98	0.08	0.55	0.66	0.95	0.80	0.80	0.80	0.45	0.99

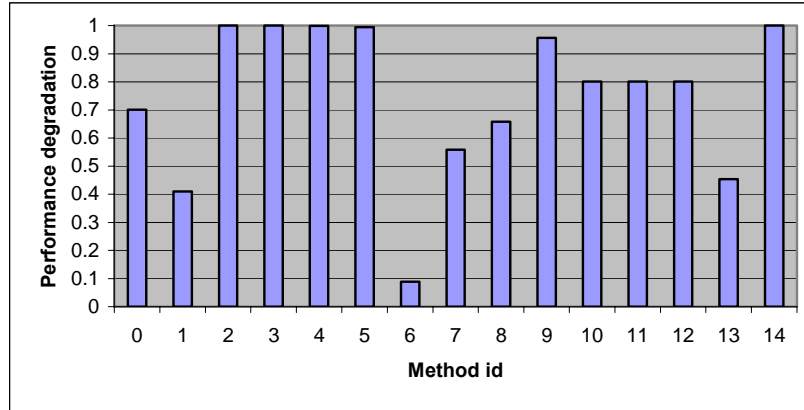


Figure 3. Performance Degradation ( $T_{\text{fault-free}}/T_{\text{faulty}}$ ) when a method is replaced by its software equivalent.

of our jpeg application show that only 24 instructions of the LEON instruction set have been used by the application. Therefore, we can remove the unused instructions from the LEON core and construct a new application-specific core which is used as a redundant module for the main processor. However, completely removing an instruction from a processor core is rather sophisticated. Moreover, removing one instruction of a data path element often has a trivial effect on the area. As a result, we found and removed some unused data paths; i.e.: multiplication, division, and floating-point unit. These data paths are not used, because in our JPEG application, multiplication, division and floating-point operations are implemented by simple shift and add operations along with lookup tables.

Table 3. The area of LEON and the removed data paths (Number of gates)

Components	Processor core(Integer and floating-point units)	Multiply Data Path	Divide Data Path	Floating-point Data Path
Area	29910	1168	1257	7477

Table 3 shows the area of the LEON processor core and the data paths which have been removed in the redundant application-specific processor core. By removing the unused data paths, the new application-specific core needs 22430 gates. This result shows that the processor can be made fault-tolerant by about 70% area (and consequently cost) overhead while standby sparing and the TMR impose 100% and 200% cost overhead to the system, respectively. As a result, since we only remove the unused data paths in the redundant processor core, the presented approach can tolerate the processor core faults, without performance loss and with less area overhead than traditional methods.

## 6. Conclusions

In this paper we presented a fault-tolerant method for object-oriented application-specific instruction-set processors. These processors are synthesized from an object-oriented high-level description by a hardware-software co-design method. We first presented a fault-tolerant approach for the functional units. In the proposed fault-tolerant approach, we first use an assertion based method to detect faults. After detecting a permanent fault, a faulty hardware functional unit is replaced by a corresponding software equivalent. For replacement, we take advantage of object-oriented polymorphism to redirect the faulty-method invocations to its software equivalent. We showed that this fault-tolerant mechanism can tolerate faults without any cost/area overhead and with acceptable performance overhead and is suitable for applications with high reliability demand under heavy cost constrains. Moreover, by using this method along with the standby sparing, we can make a trade-off between performance degradation and cost overhead. The proposed mechanism can also be used to cover design faults. Easy and automated switching using object-oriented polymorphism and low cost/area are the merits of this methodology. Moreover, we

presented a fault tolerant approach for the processor core. In this approach we design an application-specific redundant module for the processor core which consists of the subset of data paths of the processor core that has been used in the running application. This approach can tolerate the processor core faults with no performance loss and less cost overhead than traditional replication approaches.

## 7. ACKNOWLEDGMENTS

This work is supported by a research grant from the Department of High-Tech. Industries, Ministry of Industries and Mines of the Islamic Republic of Iran.

## 8. REFERENCES

- [1]B. W. Johnson. Design and analysis of fault tolerant systems. Addison-Wesley,1989.
- [2]M. Goudarzi, S. Hessabi, and A. Mycroft, "Object-Oriented Embedded System Development Based on Synthesis and Reuse of OO-ASIPs," Journal of Universal Computer

Science," vol. 10, no. 9, pp. 1123-1155, Springer-Verlag, September 2004.

[3] F. da Silva, and Alice M. Tokarnia, "Synthesis of Fault-Tolerant Embedded Systems Based on Dynamically Reconfigurable FPGAs," IPDPS'04, New Mexico - 2004

[4] R. Karri, K. Kim, and M. Potkonjak, "Computer Aided Design of Fault-Tolerant Application Specific Programmable Processors," IEEE Transactions on computers, 2000

[5] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici. "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration," fccm, p. 165, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.

[6] R. Ernst, and P. Nowotnick, "Fault Tolerant VLSI Design with Functional Block Redundancy," Proc. Int'l Conf. Computer Design, IEEE Computer Society Press, Los Alamitos, Calif., pp. 432-436, 1991.

[7] M. Pflanz, and H. Theodor Vierhaus. "Generating Reliable Embedded Processors," IEEE Micro, vol. 18, no. 5, pp. 33-41, September/October 1998.

[8] O. Hébert, I. C. Kraljic, and Y. Savaria, "A method to derive application specific embedded processing cores," Proc. of the 8<sup>th</sup>. International workshop on hardware/software co-design, USA, pp. 88-92, Sep.2000.

[9] D. V. Schuehler, B. C. Brodie, R. D. Chamberlain, R. K. Cytron, S. J. Friedman, J. Fritts, P. Jones, P. Krishnamurthy, J. W. Lockwood, S. Padmanabhan, and H. Zhang, "Microarchitecture Optimization for Embedded Systems," 8th High Performance Embedded Computing Workshop (HPEC-8), September 28, 2004.

[10] M. Goudarzi, S. Hessabi, and A. Mycroft, "No-overhead polymorphism in network-on-chip implementation of object-

oriented models," Proc. of Design Automation and Test in Europe (DATE'04), February 2004.

[11] M. Fazeli, R. Farivar, S. Hessabi, and S. G. Miremadi, "A Fault Tolerant Approach to Object Oriented Design and Synthesis of Embedded Systems," The Second Latin-American Symposium on Dependable Computing, Brazil, October 2005.

[12] LEON processor Core, <http://www.gaisler.com/leon>, September 2005.

[13] M. Levy, "Customized processors: have it your way," EDN, January 7, pp. 97-104, 1999.

[14] D. Bursky, "Tool Suite Enables Designers To Craft Customized Embedded Processors". Electronic Design, volume 47, number 3, February 8, 1999.

[15] P. Shirvani, and J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories," vts, p. 440, 1999 17<sup>th</sup>. IEEE VLSI Test Symposium, 1999.

[16] A. Agarwal, B. C. Paul, and K. Roy. "A Novel Fault Tolerant Cache to Improve Yield in Nanometer Technologies," iolts, p. 149, International On-Line Testing Symposium, 10th IEEE (IOLTS'04), 2004.

[17] A.M. Gharehbaghi, S. Hessabi, B. Hamdin Yaran, and M. Goudarzi, "Integrating Assertion-Based Verification into System-Level Synthesis Methodology", Proc. of International Conference on Microelectronics (ICM'04), Tunisia, December 2004.

[18] Synopsis synthesis tools, <http://www.synopsys.com>.