2009 Asian Test Symposium

A FPGA-based Reconfigurable Software Architecture for Highly Dependable Systems

Stefano Di Carlo, Paolo Prinetto, Alberto Scionti
Control and Computer Engineering Department
Politecnico di Torino
Torino, Italy
Email: {stefano.dicarlo,paolo.prinetto,alberto.scionti}@polito.it

***Abstract*—Nowadays, Systems-On-Chip are commonly equipped with reconfigurable hardware. The use of hybrid architectures based on a mixture of general purpose processors and reconfigurable components has gained importance across the scientific community allowing a significant improvement of computational performance. Along with the demand for performance, the great sensitivity of reconfigurable hardware devices to physical defects lead to the request of highly dependable and fault tolerant systems. This paper proposes an FPGA-based reconfigurable software architecture able to abstract the underlying hardware platform giving an homogeneous view of it. The abstraction mechanism is used to implement fault tolerance mechanisms with a minimum impact on the system performance.**

I. Introduction

Systems-on-Chip (SoC) are a major revolution in IC design where the whole functionality of a system is placed on a single chip. Presently, SoCs include several tens of million gates, multiple IP cores, and complex on-chip buses and protocols.

Most of todays virtual components (i.e., IP cores) are constituted of reconfigurable hardware [1]. Field Programmable Gate Arrays (FPGAs) represent the state-of-the-art in the field of reconfigurable hardware and are nowadays common parts of complex SoCs [2], [3], [4].

Hybrid architectures based on a mixture of general purpose processors, and reconfigurable hardware components are increasingly used to accelerate computational intensive tasks. Examples of these approaches can be found in [9], [10]. In the former, a FPGA-based coprocessor has been developed to accelerate the simulation of a molecular dynamic's application, where the movements of atoms in a substance over the time are modeled. The authors present an hardware/software approach in which the most computationally intensive tasks of the application are selected and accelerated directly on a FPGA device, while the remaining part of the application is still executed on a traditional general purpose processor. In the latter, the authors propose the implementation of a multi-objective evolutionary optimization algorithm on a reconfigurable hardware based system. Another example can be found in [11], where a FPGA-based accelerator is used to improve the performance of the process scheduler of a real-time operating system.

Together with design and performance problems, the advance of the manufacturing technology and architectures makes SoCs and reconfigurable hardware more sensible to physical defects, thus requiring new solutions to design fault tolerant systems able to continue their mission even in presence of a partial failure of the internal components [5], [6], [7], [8], [14].

This paper exploits hardware abstraction into hybrid architectures together with software reconfiguration, to enable cost-effective implementations of dependable systems. Automatic re-configurability of software is utilized to design self-healing systems capable of autonomous recovery from temporary errors and permanent faults detected both into reconfigurable hardware cores, and limited portions of general purpose microprocessors (e.g., floating point unit).

A programming framework, based on a XML description of a set of available reconfiguration mechanisms is used to provide programmers with a development kit to write programs able to automatically reconfigure selected functionalities every time a hardware block involved into the computation is detected as faulty. The identification of faulty elements, out of the scope of this paper, can be performed by means of on-line Hardware/Software Built-In-Self-Test techniques, and notified to the target programs through the operating system by means of interrupt services and Inter Process Communication (IPC) mechanisms.

This architecture allows to recover from temporary errors (e.g., errors induced by SEUs) and to repair permanent faults (e.g., stuck-at faults) with a minimum impact on the system performance. It therefore ensures very high data integrity and availability without external intervention. These capabilities make this solution useful for a variety of dependable applications including unmanned remote applications such as deep space exploration, or mission critical ones.

The paper is organized as follows: section II gives a general view of the proposed software framework, section III analyzes the internal switching mechanism adopted for program reconfiguration, while section IV proposes experimental results. Finally section V concludes the paper.

II. The framework overview

Due to their complexity, hybrid architectures require the introduction of new programming models. For the sake of simplicity, we consider in this paper a simple system composed of a general purpose microprocessor (GPP), together with

1081-7735/09 $26.00 © 2009 IEEE
DOI 10.1109/ATS.2009.53

its memory, coupled with a reconfigurable hardware module based on a FPGA component (RH). This system includes all peculiarities of more complex hybrid architectures, and therefore allows us the development of a general methodology that can be later extended to more complex designs. The hybrid architecture considered in this paper is depicted in Fig. 1.

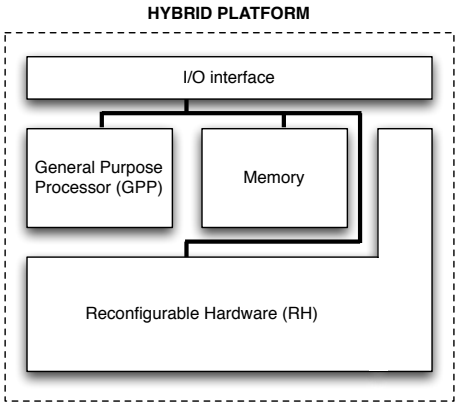

Fig. 1. Hybrid system architecture

In general the application that has to be executed on the hybrid architecture needs to be partitioned into a set of tasks. Computational intensive tasks are usually executed on RH, while the remaining ones can be executed on GPP. In order to let the programmer dealing with an homogeneous system instead of two separate entities, hardware abstraction is usually exploited [15]. In [16], Andrews et al. propose *hthreads* (or hybrid threads), an abstract computational model that actually allows thread partitioning between a general purpose processor and a reconfigurable device. It is composed of a hardware/software codesigned operating system and middleware services that support the multithreaded programming model. The hthreads compiler and run-time libraries allow programmers to write multithreaded programs with the standard C language. The hthreads operating system and middleware services provide the mechanisms that allow the threads to run on either the general purpose microprocessor, or within a custom circuit on the FPGA. In the hthreads design flow, programmers express their system computations using traditional pthreads semantics. The main drawback of this solution is the rigid distinction between the portion of the application executed by specialized hardware, and the one executed by the general purpose microprocessor. In order to efficiently exploit software reconfiguration for implementing fault tolerance systems, software applications should be able to dynamically map the execution of different functionalities both on the general purpose hardware, and on the reconfigurable hardware. This in turns requires providing the application itself with a structured description of the available reconfiguration facilities that can be exploited at run-time to reconfigure the computational tasks every time a fault is detected.

Software Based Self-Test (SBST) techniques executed on GPP, as well as embedded hardware Built-In Self-Test (BIST) facilities directly embedded into the hardware cores mapped on RH are used to check the correct behavior of the different hardware blocks. A monitor, either implemented as a hardware component or a software routine, is in charge of collecting test responses and generating proper reconfiguration events into the system. The time required for the test execution and the system reconfiguration has a limited impact on the overall performance.

Figure 2 shows the structure of the software framework we propose that can be logically split into two main parts: (i) the *exploitation package*, and (ii) the *software support package*.

The exploitation package acts as a middleware layer, exporting software modules used to manage the underlying hardware platform. In particular it exports information concerning the hardware and software facilities available at the operating system level. This information can be used by a software component (or just by the operating system itself) as a database of available reconfiguration alternatives, allowing to optimally decide how to map application functionalities. From the reliability point of view this allows to take optimal decision at run-time on how to replace faulty hardware functions on RH or faulty units on GPP.

The software reconfiguration is based on an automatic switching mechanism: when a hardware failure is detected, a notification is sent through the operating system to the program that, based on the available replacement facilities, can eventually replace the faulty functionality with a different hardware implementation, or with an equivalent software version, executed on GPP. Similarly, if one of the software functions can not be correctly executed due to a hardware fault in GPP (e.g., a fault in the FPU), it can be replaced by an equivalent hardware function.

The software support package contains software elements (i.e., a software library and the integrated development environment) used to realize the hardware abstraction mechanism. It provides the designer with a transparent mechanism to access both software and hardware resources using a uniform interface, thus giving a flexible way to split the application.

A. Exploitation package

The exploitation package resorts to four basic elements to provide hardware virtualization at the application level: (i) the *hardware configuration files*, (ii) the *operating system drivers*, (iii) the *function files*, and (iv) the *description file*.

A hardware configuration file identifies a hardware component that can be mapped into RH to perform a certain function. FPGA devices, representing our target reconfigurable components, can be configured by mean of a binary bitstream file containing the mapping of the internal configuration facilities. A library of these files can be therefore stored to form a repository of available hardware functions. Each core should be eventually provided with an embedded test mechanism and a monitor block able to check the correct behavior of the core itself, and to notify faulty conditions. In order to have a general architecture, all available blocks should be provided with a common access interface, e.g., a register file used to configure

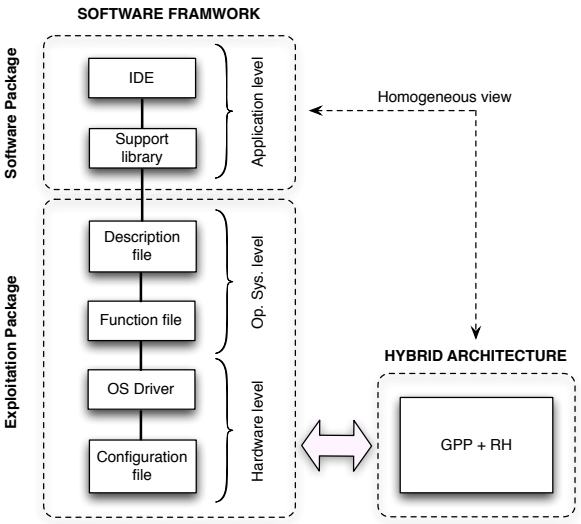

Fig. 2. The proposed high dependable and fault tolerant framework.

the core with a set of specific parameters, or to read back the result of the computation.

In order to decouple the hardware layer from the different software layers, the actual communication with the hardware cores should be managed through a dedicated operating system driver provided together with each core. The driver is also in charge of collecting hardware notifications of faulty conditions, and generating proper notifications to the programs currently using the faulty cores. The driver may also issue reconfiguration requests to optimally balance the system load.

All available functionalities, both at the hardware level and at the software one, are actually exported to the program through a set of function files described using a target high level programming language. For example considering the ANSI C language [17], [18], the set of available functionalities is declared with a couple of files, one for the header of the functions, and the other one for the specific implementation. Pure software functionalities will be directly described in these function files, while hardware implemented functionalities will simply consist at this level of a set of calls to specific operating system driver functions.

Finally, the description file is used to provide a highly structured model of the available functionalities. The description file is the main component of the exploitation package. It is used to abstract the underlying hardware architecture. It is described using a high level structured description language such as the standard XML language [19]. The structure of the file can be easily navigated by a software module and can be used as a database containing the description of the available resources. Each resource (i.e software, or hardware function) is described in terms of access mechanism, performance, and location within the software framework. Fig. 3 shows an example of the internal structure of the description file for a hardware function. The access mechanism is described through the declaration of the input parameters required to correctly execute the specific function and the output parameters used to store the result of the computation. For each parameter the type is provided. The performance is described in terms of estimated execution time, which can be used to select the optimal replacement for a faulty function, while the location in the framework is given by the corresponding library that specifies the behavior of the function and the software or hardware function counterpart. A set of custom defined description parameters for each function are also available.

```
<?xml version="1.0"?>
<Target core="Core1">
    <Functions>
       <Function name="HW_function1">
          <ver>   ...  </ver>
          <owner> ...  </owner>
          <year>  ...  </year>
          <desc>  ...  </desc>
          <numberOfPars>  ...  </numberOfPars>
          <typeOfPars>  ...  </typeOfPars>
          <outputType>  ...  </outputType>
          <execTime>  ...  </execTime>
          <EqFunction>  ...  </EqFunction>
          <EqLibrary>  ...  </EqLibrary>
          <fId>  ...  </fId>
          <customField1>  ...  </customField1>
          <customField2>  ...  </customField2>
       </Function>
       <Function name="HW_function2">
          <ver>  ...  </ver>
                 ...
       </Function>
           ...
    </Functions>
</Target>
   ...
<Target core="CoreN">
   ...
```

Fig. 3. Example of the exported XML description file.

B. Software support package

The software support package provides the software designer with the possibility of writing in a simple and straightforward manner programs that can switch their execution from the hardware context to the software one, and vice versa.

In principle, it is composed of a software library and an Integrated Development Environment (IDE) (see Fig. 2). The software library contains all functions used to perform reconfiguration whenever a request occurs. These functions are used by the operating system driver to correctly handle all low level reconfiguration actions, starting from the selection of the proper component, to the bitstream configuration into RH.

The library also contains functions to access and navigate the content of the XML description file. These functions are designed to parse the content of the description file, and to collect those information that are useful for taking optimal decisions for the replacement (e.g., a faulty hardware function can be replaced with a single equivalent software function or using a set of hardware and software functions that minimize the execution time).

The IDE aims at simplifying the creation of the reconfigurable-program. The key point of this component is the possibility of writing applications as close as possible to normal software-only programs.

III. Context execution switching

In order to actively allow switching between different implementations of the same function, an efficient switching mechanism should be provided into the framework.

The mechanism we propose in this paper relies on a data structure used to store handlers of both hardware, and software versions of available functionalities, together with the capability of the operating system to notify applications when reconfiguration events coming from a faulty core are detected.

The data structure is composed of three arrays of pointers as depicted in Fig. 4. Each array element identifies a given functionality as described in the XML description file. The *hardware executable context array* stores pointers to available hardware functions while the *software executable context array* stores pointers to the software ones. A null pointer in one of these arrays indicates that the given version of the function is not available in the current implementation. Whenever multiple copies of a function are available, each element of the array is organized itself as an array of pointers. The third array (i.e., *current execution context*) contains, for each function the pointer to the currently used version, copied from one of the first two arrays.

In order to be transparent with the currently used implementation of a function, it is enough to force designers to access functionalities through pointers contained into the current execution context array. This actually provides an efficient abstraction mechanism through which accessing both GPP and RH.

Test sessions of the hardware components can be periodically scheduled during the normal behavior of the system. Whenever a fault is detected an interrupt is generated to notify the event to the operating system. When a fault is notified, the faulty function entry in the current execution context array is selected and the corresponding pointer is replaced with the one of a different hardware implementation if available or with a pure software version. The replacement is based on the result of the exploration of the available resources through the XML description file. In this way the next call to the faulty function will execute a different version not involving the faulty detected core.

The proposed mechanism assures a high level of efficiency (only few assembler instructions are generated from the compilation of the application) thanks to the fact that only a pointer copy is involved in the replacement process.

IV. Experimental platform and results

The proposed approach as been implemented into a system for encrypted data transmission. A testable AES-128 cryptographic core mapped on a reconfigurable device is used to increase the system performance. Periodic test sessions are executed on this core, and injected faults are used to simulate

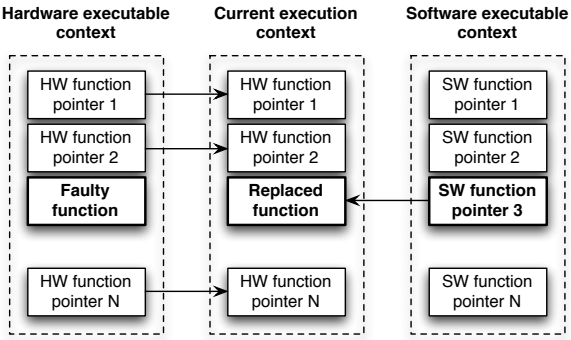

Fig. 4. Execution context switching data structure.

faulty conditions and to evaluate the recovery capability of the system.

The experiments have been performed using the ML403 Evaluation Platform equipped with a Virtex-4 FPGA™ integrating a 32-bit PowePC™ microprocessor running at 300MHz, a FPGA, and 64Kb of embedded RAM (BRAM) on the same die. The platform is equipped also with 64Mb of DDR SDRAM with a 32-bit interface, a RS232 port implemented by a Xilinx-provided core, an Ethernet communication channel, and a System ACE compact-flash-support core. Fig. 5 shows the platform used for the experimental session.

The proposed software framework has been implemented on top of the MontaVista Linux™ operating system with a 2.4.26 kernel version, specifically designed for embedded systems. The compact flash card provided together with the reference board has been split in three blocks used to store the configuration bitstream of the different FPGA components (including the AES cryptographic core), the memory swap disk partition, and the root filesystem created by means of the BusyBox [21] system. Moreover an optimized version of the operating system kernel has been compiled after the configuration of the board.

The AES encryption standard has been mapped on the reconfigurable area of the FPGA device. This core is based on the testable architecture proposed in [20] where an additional SELF-TEST operating mode is used to detect faults into the core. At the same time, an equivalent software implementation of the AES cryptographic function has been adopted, recurring to a standard cryptographic library written in C language. For the experiments the core has been designed to allow fault injection by writing a specific fault injection register. Injections are directly issued by the operating system driver whenever a user signal *sigusr1* is generated.

Both software and hardware cryptographic functions have been integrated into a single application, recurring to the proposed software abstraction mechanism. The application performs a loop that continuously executes the encryption of 128 bit data blocks. Whenever the application accesses to the cryptographic functions a test session is performed before executing the function, checking the presence of faults in the core. Fig. 7 shows the execution time of the encryption

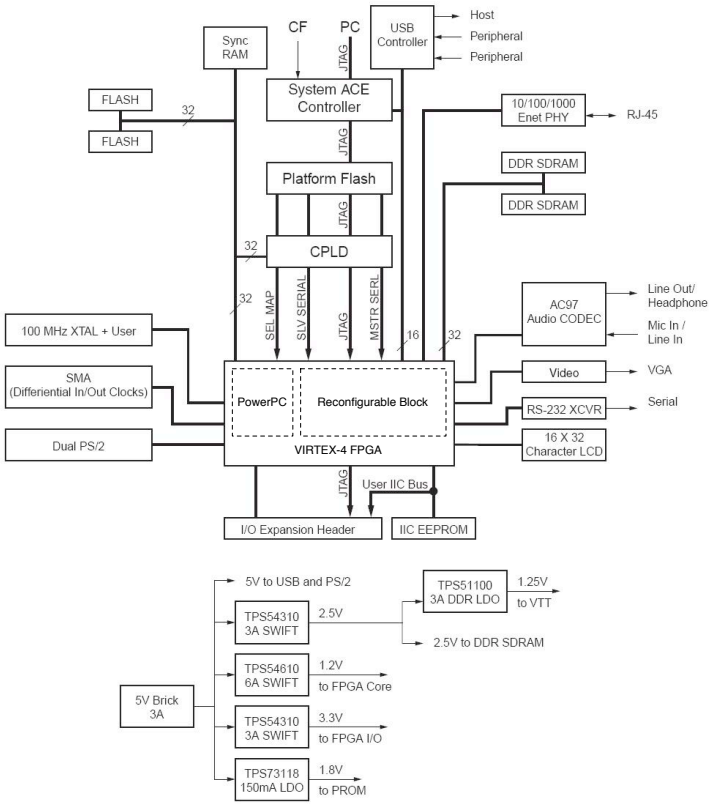

Fig. 5. The block diagram of the ML403 Evaluation platform

function before and after the reconfiguration from hardware to software. The x-axis displays the iteration number while the y-axis the function execution time. Fig. 6 shows the XML file used to export the information related to the encryption functionality of the hardware core and that of the software library.

During the first part of the execution the system is fault free and the encryption process is efficiently performed using the hardware AES cryptographic core. The hardware encryption phase takes an average time of $4.0\mu s$ oscillating between $3.0\mu s$ and $7.0\mu s$ due to the system workload.

At about 12010 iterations, a fault is detected and the software is reconfigured. This is clearly visible in Fig. 7 as there is a peak in the execution time equal to $1750.0\mu s$, caused by the detection of the fault and by the reconfiguration of the system.

After the reconfiguration phase, however, the system performance increases again and the system continues to perform its functionality, even if with reduced capabilities. The performance measured with the pure software execution is two order of magnitude slower than the one measured with the acceleration enabled (i.e., the software encryption has been performed with an average time equals to $700.0\mu s$).

V. Conclusion

In this paper we presented the implementation of a software framework for the abstraction of the different hardware resources available in a hybrid architecture that can be exploited both to design high performance software applications, and to ensure fault tolerance. A switching mechanism is provided as a recovery method after the detection of a fault . The approach allows the implementation of a highly dependable system with a minimum impact on the performance, and without the interruption of the executed task.

Acknowledgment

The authors would like to thank Matteo Bosio for the help in setting up the experimentation platform.

References

[1] D. Bouldin, *Platform-Based System-on-Chip Design*, Proceedings of 2003 Microelectronic Systems Education Conference (MSE), Anaheim, CA, pp. 48-49, June 1-2, 2003.

[2] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis,*Performance improvements using coarse- grain reconfigurable logic in embedded SOCs*, Field Programmable Logic and Applications, 2005. International Conference on Volume, Issue, 24-26 Aug. 2005 Page(s): 630 - 635.

[3] J. Villareal et al, *Improving Software Performance with Configurable Logic*, in Design Automation for Embedded Systems (DAES), vol. 7, pp. 325-339, 2002.

```
<?xml version="1.0"?>
<Target core="CryptoCore">
    <Functions>
       <Function name="Encryption_data">
          <ver> 1.0 </ver>
          <owner> Politecnico di Torino </owner>
          <year> 2009 </year>
          <desc> AES encryption function </desc>
          <numberOfPars> 4 </numberOfPars>
          <typeOfPars> int,int,int,int </typeOfPars>
          <outputType> int </outputType>
          <execTime> 3.0us </execTime>
          <EqFunction> encryption_data </EqFunction>
          <EqLibrary> aeslib </EqLibrary>
          <fId> 1 </fId>
      </Function>
    </Functions>
</Target>
<Target core="General Purpose Processor">
    <Functions>
      <Function name="encryption_data">
          <ver> 1.0 </ver>
          <owner> Politecnico di Torino </owner>
          <year> 2009 </year>
          <desc> AES encryption function </desc>
          <numberOfPars> 4 </numberOfPars>
          <typeOfPars> int,int,int,int </typeOfPars>
          <outputType> int </outputType>
          <execTime> 690.0us </execTime>
          <EqFunction> Encryption_data </EqFunction>
          <EqLibrary> aeslib </EqLibrary>
          <fId> 2 </fId>
      </Function>
    </Functions>
</Target>
```

Fig. 6. The XML description file for the experimental platform.

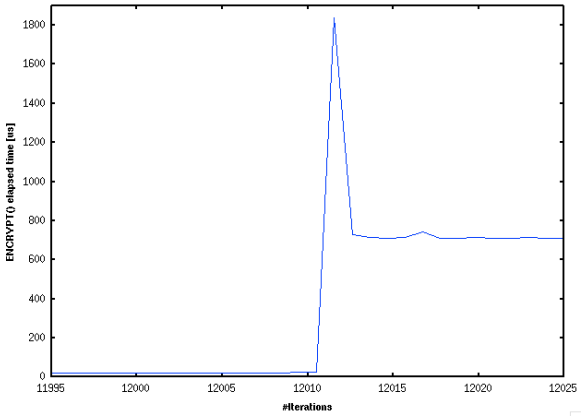

Fig. 7. AES encryption time response

[4] G. Stitt et al., *Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems*, in ACM Trans. on Embedded Computing Systems (TECS), vol.3, no.1, pp. 218-232, Feb. 2004.

[5] I. G. Harris, P. R. Menon, R. Tessier, *BIST-based delay path testing in FPGA architectures*, Test Conference, 2001. Proceedings. International, Volume, Issue, 2001 Page(s):932 - 938.

[6] F. Hanchek and S. Dutt, *Methodologies for tolerating cell and interconnect faults in FPGAs*, Computers, IEEE Transactions on, Volume 47, Issue 1, Jan 1998 Page(s):15 - 33.

[7] Y. Shu-Yi and E. J. McCluskey *Permanent fault repair for FPGAs with limited redundant area*, Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on Volume , Issue , 2001 Page(s):125 - 133.

[8] N. Campregher, *FPGA interconnect fault tolerance*, Field Programmable Logic and Applications, 2005. International Conference on, Volume, Issue, 24-26 Aug. 2005 Page(s): 725 - 726.

[9] R. Scrofano, M. B. Gokhale, F. Trouw and V. K. Prasanna, *Accelerating Molecular Dynamics Simulations with Reconfigurable Computers*, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 19, NO. 6, JUNE 2008.

[10] S. Bonissone and R. Subbu, *Evolutionary Multiobjective Optimization on a Chip*, Proceedings of the 2007 IEEE Workshop on Evolvable and Adaptive Hardware (WEAH 2007).

[11] M. Song, S. H. Hong and Y.Chung, *Reducing the overhead of real-time operating system through reconfigurable hardware*, Digital System Design Architectures, Methods and Tools, 2007, DSD2007. 10th Euromicro Conference on 2007.

[12] Xilinx Application Notes XAPP216, *Correcting Single-Event Upset Through Virtex Partial Reconfiguration*, 2000.

[13] M. Sonza Reorda, L. Sterpone and M. Violante, *Multiple errors produced by single upsets in FPGA configuration memory: a possible solution*, IEEE European Test Symposium, 2005, pp. 136-141.

[14] M. Violante and L. Sterpone, *Hardening FPGA-based systems against SEUs: A new design methodology*, JOURNAL OF COMPUTERS, VOL. 1, NO. 1, APRIL 2006.

[15] D. Andrews, D. Niehaus and P. Ashenden, *Programming models for hybrid cpu/fpga chips*, IEEE Computer, 37(1):118-120, January 2004.

[16] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot and E. Komp, *Achieving programming model abstractions for reconfigurable computing*, Very Large Scale Integration (VLSI) Systems, IEEE Transaction on, 16(1):34-44, January 2008.

[17] The standard ANSI C language: *http://www.open-std.org/jtc1/sc22/wg14/.*

[18] B. W. Kernighan and D M. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall, Inc., 1988, ISBN 0-13-110370-9.

[19] The standard XML language: *http://www.w3.org/XML/.*

[20] G. Di Natale, M. Doulcier, M. L. Flotte and B. Rouzeyre, *Self-Test techniques for crypto-devices*, Very Large Scale Integration (VLSI) Systems, IEEE Transaction on, 2008.

[21] The BusyBox system: *http://www.busybox.net/about.html*