Multi FPGA Based Novel Reconfigurable Hybrid Architecture for High Performance Computing

Sunil Kr. Singh[1], R. K. Singh[2], M.P.S. Bhatia[3], Ratnakar Madan[4]

[1] Ph.D, Research scholar, Uttarakhand Technical University, Uttarakhand, INDIA

[2] Professor, Uttarakhand Technical University,Dehradun, Uttarakhand, INDIA,

[3]Professor, CSE Department, NSIT, University of Delhi, New Delhi, INDIA

[4] UG Research scholar, Bharati Vidyapeeth College of Engineering, New Delhi, INDIA

Abstract

The growth of the verticals depending on the reconfigurable computing has been very fast. S atellite systems, land rovers, rocket launchers and other heavy duty high performance systems are making use of reconfigurable processors. However, still these processors are not able to provide for the strict hard real time deadlines required. The reason behind is the flexibility of being reconfigured, the delay in the transfer of signals and the time required to reconfigure the part of FPGA based multiprocessors is slightly higher. Thus we are proposing a Multi FPGA based Novel Reconfigurable hybrid architecture which provides for a l esser delay, more reliability and a higher throughput. This system architecture has been developed with the intent of reducing the dynamic decision making so as to reduce the run time and also by minimising the number of context switching operations by providing more than one FPGA processors. So that the need for context switching in normal circumstances is reduced to zero and is only required in case a failure occurs in the system.

Keywords: *Reconfigurable Computing, FPGA, Hybrid Architecture, RPU, Context switching*

I. Introduction:

In the past few years, applications requiring high performance computing have become really heavy and need lot of computations to be done. To solve this problem, a lot of research has been going on in the area of reconfigurable computing, where the hardware is reconfigured at runtime to adapt to the need of the applications. Currently, the products that are available in the market are a combination of a host, General Purpose Processor (GPU) and a R econfigurable Processor (RPU) on a single VLSI chip. The performance (in terms of time delays) of this architecture is acceptable till the application demands more of software based processing but when the application needs more hardware processing, this architecture fails to deliver the required performance because the GPU present cannot be used for hardware tasks, it can only be used for software tasks [11]. So whenever high percentage of hardware processing is required it leads to increase in the number of the times the RPU is reconfigured by the GPU. During this “reconfiguration time” no other work is done by the GPU and the RPU. This increases the delay and also the turnaround time.

The delay time in hardware intensive tasks is also increased due to a l arge number of context switches are required in such systems. The increase in number of context switches is due to limited number of hardware functional units. As the number of hardware intensive tasks increase, the number of context switches increase proportionately. Higher the number of context switches, more is the time the GPU spends doing no effective work. This further has a cas cading impact and increases the turnaround time of the software intensive tasks. The cumulative increase in delay and turnaround time for both hardware intensive tasks and software intensive tasks is an area of concern.

It has been seen that the applications like weather forecasting, Remote sensing etc. that make the use of supercomputers have more of hardware dependant processing and vector processing requirements [4]. In future more such applications are expected to emerge and hence a better approach would be required to cater to such needs. It has been shown that the current “hybrid architecture” is capable of handling the present requirements, but a more dependable and sophisticated hybrid architecture is necessary for future requirements [10].

In this contribution to the world of reconfigurable computing, we propose a Multi FPGA based (more than 1 FPGA based reconfigurable processor) Hybrid architecture, without the separate General Purpose processor that is being commonly employed in the present products available from the leading manufacturers like Altera & Xiline. In our proposed hybrid architecture we configure a portion of the RPU to act as the GPU. Although this increases the delay time for software intensive tasks but the decrease in throughput time for hardware intensive tasks is considerable and this is where we intend to use our proposed hybrid architecture. Through the use of an extended operating system with a real time kernel we reconfigure the RPUs according to the application requirement and take full leverage of the multi-FPGA environment. The extended OS Kernel efficiently reconfigures the processors whenever required, switches between the multiple RPUs that are being used and also provides for message passing and inter-process communication between the processes running on s ame RPU and also between processes running on different RPUs. The architecture that we propose, not only compensates the removal of GPU, but also aims to reduce the power consumption by reducing the power being wasted in the present "Hybrid Architecture" during the reconfiguration time and also during normal hardware processing when the GPU sits idle and consumes power. Not only we aim to save power, our aim is to further enhance the performance and reduce the turnaround time by reducing the waiting time and also by extending the multiprocessing terminology to Multiple Reconfigurable Processors.

II. Current Hybrid Architecture & Related Operating System:

Hybrid architecture is a co mbination of a G PU and RPU (FPGA or CPLD based). In this paper we refer to the system composed of a general purpose microprocessor (GPP), together with its memory, coupled with a reconfigurable hardware module based on a FPGA component (RH), as we take it as a b ase architecture and modify it to propose our Novel architecture.

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 for modifications to be made to lead to our novel Hybrid. In general, any 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 a homogeneous system instead of two separate entities, hardware abstraction is usually exploited [2]. In propose hthreads (or hybrid threads), an abstract computational mode that actually allows thread partitioning between a general purpose processor and a reconfigurable device [13][2]. It is composed of a hardware/software co-designed 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.

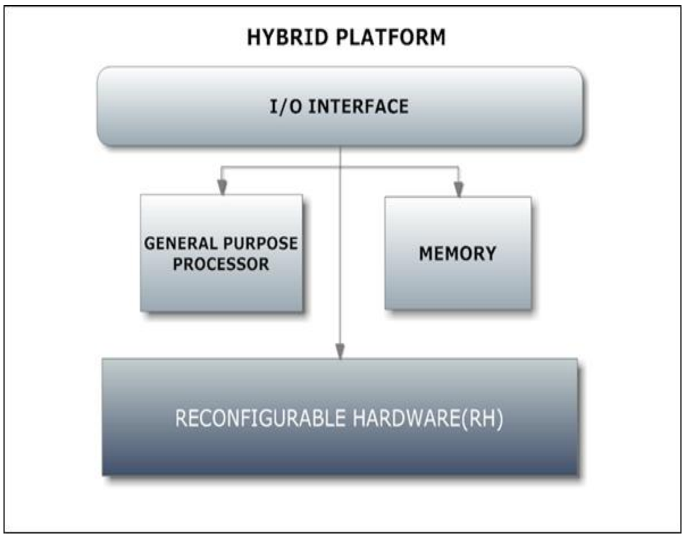


Figure 1. Classical Hybrid Architecture using Reconfigurable Hardware

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 are now 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 behaviour of the different hardware blocks [5]. 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 l imited impact on the overall performance.

Figure 2 shows the structure of the software framework, 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 deciding 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 h ardware 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.

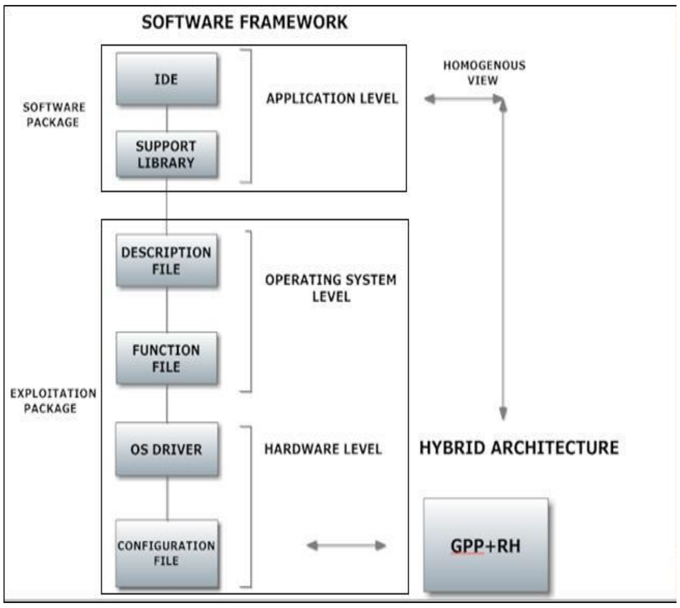


Figure 2. The classical dependable and fault tolerant framework

Similarly, if one of the software functions cannot 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 [6][7].

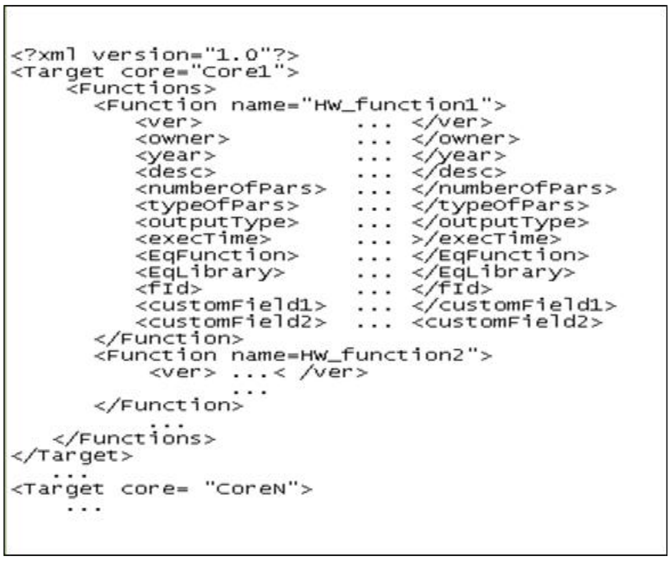
➢ Exploitation package

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 the target reconfigurable components, which can be configured by mean of a b inary bit stream file containing the mapping of the internal configuration facilities.

A library of these files is stored to form a repository of available hardware functions. Each core is eventually provided with an embedded test mechanism and a monitor block able to check the correct behaviour of the core itself, and to notify faulty conditions. In order to have a general architecture, all available blocks are provided with a common access interface, e.g., a register file used to configure 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 is managed through a d edicated 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 also issues 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 [1][12], 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 are directly described in these function files, while hardware implemented functionalities 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 (In standard XML lang.). The structure of the file is easily navigated by a software module and used as a d atabase 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 behaviour of the function and the software or hardware function counterpart.

➢ 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 s oftware library and an Integrated Development Environment (IDE) (see Fig. 2).



```xml
<?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">
    ...
```

Figure 3. Example of the exported XML description file

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 bit stream 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 that information that are useful for taking optimal decisions for the replacement (e.g., a f aulty hardware function can be replaced with a single equivalent software function or using a s et 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.

We take this hybrid architecture proposed by Stefano Di Carlo et al as the basis of our novel architecture presented ahead, we also propose the extension of the Operating System used to manage the changes in the architecture and present an efficient static efficient functional unit mapping algorithm for the multiple reconfigurable processors taking into account various factors which provide for fast execution, reduction in power usage , and also a reduction in the heat generated reducing the extensive need of cooling functional units required.

III. Proposed Hardware Architecture & the Extended Operating System:

In our proposed Hybrid architecture (Figure 4), we do away with a separate GPU being used currently, and instead add a Reconfigurable Processor (RPU2). With the GPU functionality being incorporated in the partially pre – configured RPU1. There have been free blocks left in the RPU1 for future use. This has been done to manage any hardware problem that may occur in the GPU part. The block reserved for the future use can then be partially reconfigured during runtime and that functionality can be mapped to this hardware block [3][14]. Both the RPUs and the Input Output Interface have been connected to the memory hierarchy of the system. The kind of memory systems used, depends extensively on the applications to be used, hence we have not specified the hierarchy in detail.

As in the current architecture, any application is first split into tasks, computationally intensive will be executed on RH while the remaining ones on the GPP. We also propose for the same hardware abstraction that is being currently exploited but we follow the hthreads, the abstract computation mode instead of being provide the flexibility for interchanging of thread execution on the GPP or the RH at the runtime. This is done as we have provided for free blocks which can be reconfigured by the bitstream file of the faulty block in the GPU and the functionality can be replicated. The Hardware Description Repository files maintained in the memory are updated as soon as a fault is detected in any of the blocks, and also when the execution of a task is completed in the block.

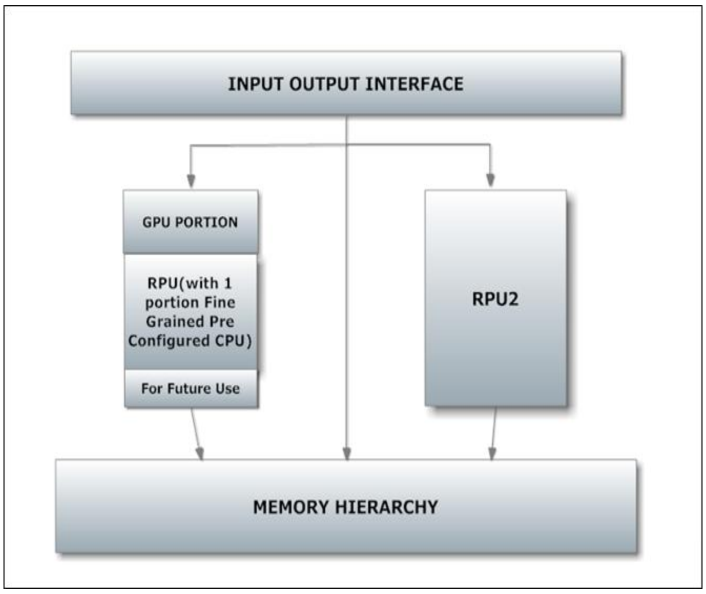


Figure 4. The proposed novel hybrid high fault tolerant architecture

The files in our case, also store the time (t_l) when the execution of the last task is completed. This factor plays an important part when a block is selected for the task to be performed. We avoid the continuous using of one block to evenly distribute the utilization of the block so as to avoid generating excess heat from one block. We use the LRU algorithm to choose which block executes a particular task. However, if failure in the GPU portion occurs during the execution of a task, a failure notification is sent to the application through the Operating System. This task is then, allowed to execute on the RH, while the functionality of the faulty GPU block is replicated in the "future use" block [9].

A task that has been structured by the application to be run on the RH, will be executed on the RH part. Either of the RPU s could be chosen for hardware tasks. The LRU algorithm maps the task to the selected block in the RPU at the time of software compilation. The mapping decision also considers the factors such as bus usage for e.g. if the RPU2- memory bus is in use, and the task is high priority or needs a near real time completion, then it is mapped to RPU1 and vice versa. This can be indicated by the developer during the time of system application development by setting the parameters provided in the IDE.

All the above decisions are taken by the components of the hardware\software co-designed extended operating systems giving the programmer's choice of parameters a p riority. The extended operating system provides the same hardware abstraction as in the current architecture, the hthreads model based on the pthreads semantics mentioned above can be used. This means that the application developers do n ot have to change the application and the development process remains the same. Although the flexibility of the hthreads model is considered a drawback in the current architecture, we have overcome that drawback by providing ample hardware and computation resources, which helps in fast execution as the decision of which processing element to choose is not made at the run time rather during the development of the application itself. We also keep the "fault tolerant" property of the current architecture intact, by mapping the hardware block dynamically for those tasks meant to be executed by the RH and providing for execution of the GPP tasks on the RH part if a runtime failure in the incorporated GPU is detected.

All in all the framework of the current architecture is maintained as shown in Figure2, only the components of the framework, such as the Hybrid architecture have been completely modified and the "exploitation package " in the current framework has been extended to manage the multi RPU architecture and also for limiting the number of context switches. The "Software package" has been slightly modified such that the decision to choose between the GPP and the RH now rests with the developer and has to be taken at the development time, however in case of a failure, the GPP tasks are automatically mapped to RH making the use of the extensive "exploitation package" in the proposed extended operating system.

VI. Context Switching:

A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) from one process or thread to another. A process (also sometimes referred to as a task) is an executing (i.e.,running) instance of a program. In Linux, threads are lightweight processes that can run in parallel and share an address space (i.e., a r ange of memory locations) and other resources with their parent processes (i.e., the processes that created them).

Context switching can be described in slightly more detail as the kernel (i.e., the core of the operating system) performing the following activities with regard to processes (including threads) on the processor: (1) suspending the progression of one process and storing the processor's state (i.e., the context) for that process somewhere in memory, (2) retrieving the context of the next process from memory and restoring it in the processor's registers and (3) returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

A context switch is sometimes described as the kernel suspending execution of one process on the CPU and resuming execution of some other process that had previously been suspended. Although this wording can help clarify the concept, it can be confusing in itself because a process is, by definition, an executing instance of a program. Thus the wording suspending progression of a process might be preferable.

A context switching processor although increases the flexibility by providing a multi-programmed environment but the context switching is also a burden for the processor as during the time which the processor is busy in performing a context switch, it is not doing any effective work. This flexibility may cost a heavy price in a real time environment where hard time deadline and precision is required [8]. The basic intent of creating a multi FPGA architecture is to minimize context switching. Since we have dedicated functional units, predefined for each of the tasks at the compile time, there are no chances of synchronization or a collision problem. The only two possibilities when a context switch would be required in a multi FPGA architecture where there is a r eal time constraint and one of the mapped hardware functional unit fails. In this particular case, the contents of the registers would have to be saved and loaded to the new functional backup unit predefined at the compile time. The mapping for the backup units can be done using a normalized many to many function.

The second case for the context switching arises when a number of copies of the same process are waiting to be executed. In this scenario, the application developer will have a ch oice while compiling the code for a multithreaded program. If the system is hard real time, a separate hardware unit will be allocated for each copy of the thread, while if the system is soft real time, developer has a choice of marking the thread as a software intensive thread with higher priority as compared to the normal software intensive threads. This works fine for a system which has a soft real time constraint.

The GPP component performs the activity of context switch whenever a fault is detected in any of the hardware functional units. The activity of context switching has been mapped to the GPP because the context switch requires saving the content of the registers and then loading the registers with the new values. This process would be better executed if it is mapped for software execution. However, taking into consideration the importance of the process of context switching, a dedicated processing unit in RPU1 is provided as a backup unit for context switching. In case a problem occurs in the GPU portion, the context switching processing is mapped to the hardware on a high priority bases to prevent any errors and delay in the mechanism for context switching.

Thus, with the fixed mapping done at the compile time for software threads to hardware functional units, although we have decreased the flexibility but the performance is enhanced as there is no time wasted for continuous context switching which is a common scenario in the general purpose processors and other processors used in non real time environment.

V. Conclusion:

The use of reconfigurable processors is fast moving into the real time domain, as the technologies used in designing these processors has developed at a very fast pace and reduced the delay that use to occur while the signals were transmitted through inbuilt busses. The use of reconfigurable processors is being made in systems where a processing failure is not tolerable as the cost of the entire system is very high. The use of multi reconfigurable processors in the architecture would further give the opportunity to increase the throughput and make use of such architecture in a strict hard real time environment.

The increase in throughput in our proposed architecture has been due to two major factors. One being of mapping the threads to be executed at compile time to a dedicated hardware functional unit and separation of threads as software based and hardware based at compile time. This eliminates the need of deciding at the run-time the type of service (hardware intensive or software intensive) a task would be requiring and if hardware intensive then to which particular functional unit it should be mapped.

The second factor which contributes to faster performance is the limiting the number of context switches required. This decision to limit the number of context switches is based on the strong reason that our architecture has dedicated functional unit for each of the hardware intensive task decided at the run-time. Thus there is no chance of a collision or synchronization problem. For situations when even a high priority task needs to execute, there would be no need of a context switch as there would be a dedicated functional unit for that particular task.

The extended operating system proposed also takes into account various other factors like time (T1) when a hardware functional unit was used last time. The mapping of tasks to hardware functional units is done such that all the units are used evenly and the distribution of usage is such that the heat generated is even which helps in decreasing the capacity and cost of cooling solutions required to keep the temperature of the processing units under control.

Thus the proposed architecture provides a considerable increase in performance and reliability of the system, reducing the chances of failure of the complete systems and proving to be of immense use in high cost systems whose repair or maintenance is not possible as there is very little human interference possible after the systems have been deployed.

References

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

[2] D. Andrews, D. Niehaus and P. Ashenden,2008, Programming models for hybrid cpu/fpga chips, IEEE Computer, 37(1):118-120, January 2004.Systems, IEEE Transaction on, 16(1), page 34-44.

[3] F. Hanchek and S. Dutt, 1998, Methodologies for tolerating cell and interconnect faults in FPGAs, Computers, IEEE Transactions on, Volume 47, Issue 1, Page 15 - 33.

[4] G. Stitt et al., 2004, 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.

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

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

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

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

[9] N. Campregher, 2005, FPGA interconnect fault tolerance, Field Programmable Logic and Applications, International Conference on, Volume, Issue, 24-26, Page 725 - 726.

[10] R. Scrofano, M. B . Gokhale, F. Trouw and V. K. Prasanna, 2006, Accelerating Molecular Dynamics Simulations with Reconfigurable Computers, IEEE transactions on parallel and distributed systems, Vol. 19. No. 06.

[11] Sunil Kr. Singh, Ratnakar Madan, Nitisha Jain, 2011, "Reconfigurable Hybrid Architectures for High performance, Reliable embedded system" proceeding of International conference on FTICT, RKGIT, Ghaziabad, India.

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

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

[14] Y. Shu-Yi and E. J. McCluskey, 2001, Permanent fault repair for FPGAs with limited redundant area, Defect and Fault Tolerance in VLSI Systems, Proceedings 2001 IEEE International Symposium, Page 125 - 133.