ReSP: a design and validation tool for data systems

Giovanni Beltrame
European Space Agency
Keplerlaan 1
Noordwijk, NL
Giovanni.Beltrame@esa.int

Luca Fossati
European Space Agency
Keplerlaan 1
Noordwijk, NL
Luca.Fossati@esa.int

1. INTRODUCTION

The continuous increase of transistor density on a single die is leading towards the production of more and more complex systems on a single chip, with an increasing number of components. This brought to the introduction of the System-On-Chip (SoC) architecture, that integrates on a single medium all the components of a full system. However, power and heat dissipation, difficulties in increasing the clock frequency, and the need for technology reuse to reduce time-to-market push towards different solutions from the classic single-core or custom technology. A solution that is gaining widespread momentum consists in exploiting the inherent parallelism of applications, by executing them on multiple off-the-shelf processor cores. Having separate cores on a single chip reduces hot-spot brought to the definition of Multi-Processor Systems-on-Chip (MPSoCs). The design of MPSoCs raises new challenges due to the large design space and tight design and time-to-market constraints. MPSoCs are complex devices, and therefore they require some particular modelling techniques that are able to hide their inherent complexity. Nevertheless, the model has to be accurate enough to describe the entire system throughout the phases of its development, and has to provide enough flexibility to be refined iteratively up to the point where the actual device can be produced using current process technology. MPSoC technology is gaining widespread momentum in the commercial world, and it is being considered as a viable alternative for space applications [12]

These new challenges for system architects, software and hardware designers, verification specialists and system integrators may best be met by revisions to old tools, by using methods to deal with MPSoC complexities, by introducing new tools and methods working at the same abstraction levels and by moving up in abstraction to take advantage of new design approaches. Moreover, in order to obey to tight market constraints, the SoC design process must rely on pre-designed or third party components. Components obtained from different providers, and even those designed by different teams of the same company, may be heterogeneous on several aspects: design domains, interfaces, abstraction levels, granularity, etc. Therefore, component integration is required at system level.

In this context, when combining independently designed modules, the enhancement and assessment of reliability becomes particularly important; for instance specific approaches are required in order to be able to apply fault detection and fault tolerance techniques from the initial steps of the design flow and also to evaluate the effects of faults in a component while interacting with the other ones composing the MPSoC. These reliability issues are becoming more and more relevant, as the incidence of soft errors grows also at ground level [6]. Such errors are caused by radiation, and they temporarily affect memory elements so that their content may be corrupted; this situation, particularly hazardous in safety-critical systems, it is serious in general, especially when considering the embedded systems' pervasiveness in today's life.

The ones listed so far are not the only problems related to the design of the most recent embedded systems (the new MPSoC architectures in particular). Since the lifetime of a design is becoming shorter than ever before, it is very important to reduce development time as much as possible in order to fit the design in a narrow time-to-market window. On the other hand, the market is putting extreme demands on system architects to deliver high performance, low power solutions. Such conflicting demands of short time-to-market and high performance can only be met by incorporating reusability and hence, flexibility in the overall design. Since software design is more flexible than hardware design and since its errors can be corrected more easily even in later development stages, an increasing amount of system functionalities is being implemented in software. In order to be able to exploit the potential of a Multi-Processor System-on-Chip (MPSoC) architecture, this software has to be concurrent.

Unfortunately, parallel software developers must contend with problems not encountered in sequential programming; namely: non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, fault tolerance, heterogeneity, shared or distributed memory, deadlocks, and race conditions. These factors increase the difficulty of designing and debugging software applications: program sections that are individually correct and error-free may create unpredictable results when running concurrently. Moreover, the execution of a program may not be consistent.

Sometimes a program will run to completion, as expected. In other cases, the same program may unexpectedly crash, even on inputs that had been previously successfully tested. This is often referred to as the non-determinacy problem.

Attempting to debug such programs can be frustrating as the introduction of debugging statements in the code can change the behaviour of the program enough to prevent the bugs from appearing. The execution of debugging statements introduced in parallel tasks slows these tasks down, affecting their execution in different ways. This in turn makes them interact differently over time. Such a disruption is sometimes sufficient to prevent a spurious bug from showing up when the program is being debugged. This is normally called the "probe effect" [10], i.e. any attempt to observe the behaviour of a parallel system may change the behaviour of that system. The use of traditional standard interactive debuggers, such as GDB, is not very helpful either in order to solve the problems due to code parallelization as these debuggers are, in fact, built with the purpose of debugging sequential programs, and a standard debugging cycle usually consists of repeatedly executing the program (with the aid of watches, breakpoints, etc.) until the bug is found. This assumes a deterministic execution, something not always true for concurrent software.

ReSP (Reflective Simulation Platform) is a hardware simulation platform, based on the SystemC and TLM libraries, particularly designed to address the aforementioned issues; ReSP is primarily targeted to the simulation of Multi-Processor Systems, though it can in general be used for any hardware system. In other words, ReSP provides a modelling methodology and framework for high-level co-design and co-simulation of embedded systems, addressing issues related to debugging and fault analysis.

The paper is organized as follows: Section 2 presents ReSP architecture and components; Sections 3 and 4 shows how it can be applied to fault injection and the debugging of the embedded operating system eCos [5]. Finally in Section 5 we draw some concluding remarks and we introduce possible future developments.

2. RESP ARCHITECTURE

The main idea behind ReSP is to give the designer an easy way to specify the architecture of a system, simulate the given configuration and perform automatic analysis (such as design space exploration or reliability assessment) on it. This work is particularly suitable for platform-based design: the use of a well-defined set of architectural elements and the design space exploration on the interconnection, number and parameters of those elements, are keys for the effectiveness of the design methodology. In the following, the terms platform and framework are used to indicate the overall ReSP architecture.

ReSP uses a formalism to describe the components and the interconnections between components of a system, as many Architecture Description Languages (ADLs) do. Components are chosen from a database of SystemC modules. In the following, we refer to the term component to describe any top-level SystemC module included into the framework's database.

The proposed framework is based on the concept of reflection [7], that allows ReSP to view and modify every C++ or SystemC element (variable, method, etc.) specified in any component. In order to enable the framework's reflective capabilities, SystemC code is directly parsed and the interface files (here called wrappers) are automatically generated. This means that standard SystemC TLM IPs can be integrated in the system with minimum effort.

Figure 1 shows the overall structure of ReSP; it is clear that the system is composed of three main parts: Core, IPs and Tools. The Core is the simulator itself, which both controls the SystemC kernel and the instantiation and interconnections among the components; it is partly written in C++ and partly in Python. The IPs are the component models which are used to compose the architectures we simulate; they are written in C++ and are based on the SystemC and TLM libraries. Finally, the Tools help performing analyses and debugging on the simulated architecture.

2.1 ReSP Core

The core of the ReSP architecture is the OSCI standard SystemC kernel, as directly released by OSCI [2]. This is an advantage when compared to other works, as they require modifications to the SystemC kernel. ReSP provides a wrapper for the Python scripting language around the SystemC kernel. Python inherently supports reflection, and allows access to SystemC variables and the execution of arbitrary function calls to SystemC code. The Simulation Controller is a set of Python classes that translate commands coming from the user into SystemC function calls, controlling the simulation behaviour. As an example, it is possible to run, step, pause, or stop the simulation at runtime. Note that the pause operation is something not present natively in SystemC; this concept was introduced in [11] and it is now widely used.

The User Interface (or Human Computer Interface, HCI) is also written in Python and it represents an interface between the simulation controller and the user. This architecture allows multiple interfaces (such as command line or graphical ones) to be built. Currently we have a command line console and a socket server. The latter is used both for the

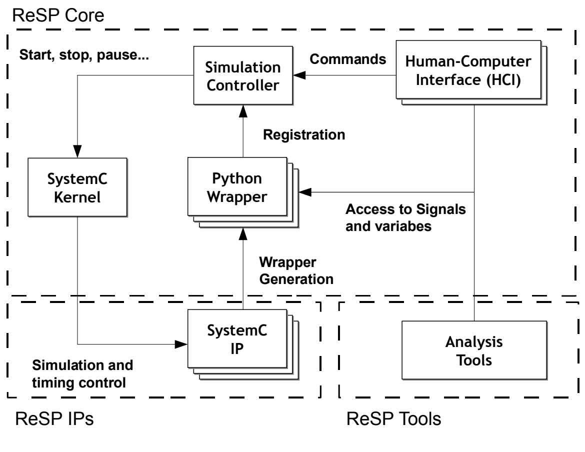

Figure 1: The architecture of the ReSP system simulation platform

integration of ReSP with external programs and also for the communication with the Graphical User Interface.

The novelty introduced by ReSP lies in the Python wrapper generation for SystemC and TLM components. ReSP deals with this step automatically, by generating the Python wrapper right after parsing the component C++ header file. The generation flow is shown in Figure 2. Each header file is parsed using GCCXML, a tool that provides an XML description of the GCC abstract syntax tree. The resulting XML description is manipulated to select all the parts that need to be exported, and then the OpenSource tool py++ [1] is used to generate Python wrapping code; this code is based on the Boost.Python library. The advantage of Boost.Python and py++ over alternative tools like SWIG (used by most other works) is that it guarantees access to all C++ declarations, even private or protected ones, through the generation of appropriate class wrappers. The Python interpreter can load the extensions generated by the ReSP flow, and have full access to the C++, and therefore SystemC, classes contained in the exported module. Another feature of the ReSP flow is that IP documentation is automatically extracted from the SystemC source code, and inserted in the Python wrapper. Python self-documentation features are then used to display such documentation through the User Interface.

2.2 ReSP IPs

One of the peculiarities of ReSP is the capability of integrating any valid SystemC component in an easy way; in fact, as described in the previous section, it is not necessary to modify components' descriptions due to the fact that ReSP automatically generates the Python wrapper. This favours external IP reusability and the description of new hardware architectures by composition of already existing components. Currently, the simulation platform includes the following component models: processors cores written using the ArchC [14] Architectural Description Language; we possess both the functional and cycle accurate versions of the PowerPC, Leon2 and ARM7 RISC processors interconnections in terms of bus and Networks-On-Chip; memory systems including simple memories and caches; miscellaneous components, such as UARTs and interrupt controllers. More components can be easily added by putting their SystemC source code in the ReSP build tree. No additional interface or glue code needs to be written, as ReSP automatically generates the appropriate component wrappers.

2.3 ReSP Tools

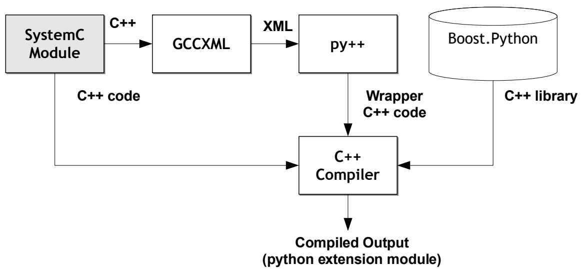

Figure 2: ReSP wrapper generation flow

The introduction of reflection paves the way for the development of a set of tools to perform system-level analysis. Any operation that requires observability, can be performed through the Python wrappers. For example, it is possible to include advanced network traffic analysis (latency, throughput, etc.) by observing the network traffic, or add power modelling to the system by extracting switching activities from the system at runtime. The biggest advantage given by the use of Python lies in the decoupling among the simulator itself and the SystemC models; the simulator does not need to have any a-priori knowledge about the components' structure: there is not need to change the simulator's code even if some components are modified.

Two execution modes are available: interactive and automatic. The first one allows step-by-step execution of the architecture under analysis. The architecture can be built using the commands exported by the User Interface: components are seen as normal Python classes which are instantiated and connected together executing standard Python commands. Automatic instantiation, by means of an XML file, is also possible. Interactive execution mode helps the designer in having a deeper insight on the modelled architecture; this mode is especially useful during debug activity which, if the reflective capabilities of the platform are also used, can be performed in a very efficient way. Automatic execution mode is used to run in batch mode a sequence of simulations.

2.4 ReSP Performance

The first experiment (Figure 3 on the left) was set-up to measure the transactional speed of the system; it consisted in the connection of basic master and slave components: the former sends characters to the latter component. On the right there are the results of the execution of a full architecture; this was created by connecting a functional Leon2 processor model (created using the ArchC [14] architectural description language) and the TLM Programmer's View (PV) memory and bus. The number of instructions per second, obtained both using native execution and execution inside ReSP, are shown.

From Figure 3 it is clear that the small performance penalty due to the additional software layer introduced by Python is negligible, especially if the advantages coming with the introduction of Python are considered.

All the experiments were hosted on a 2 GHz Intel Core 2 Duo System with 2 GB of RAM running Gentoo Linux.

3. CASE STUDY: FAULT ANALYSIS

The reflective capabilities provided in ReSP can be used also for other purposes besides architecture composition and its dynamic management; in particular, we have exploited these features to implement a fault injection environment. We have followed the SoftWare-Implemented Hardware Fault Injection (SWIFI) [3] approach, based on the modification of the components' internal state and on the simulation of the system behavior in presence of hardware failures.

Works proposed in literature pursue fault injection by means of code instrumentation for accessing the internal state of the architecture [8]. By exploiting reflection instrumentation is

Table 1: Reliability analysis: experimental results

					Error	
Application	Register	Faults	No Error	HW Detected	SW Detected	Not detected
ELPF	Reg. Bank	2000	1787	51	152	10
	PC Reg.	1000	775	12	207	6
	Other Regs	600	591	0	9	0
FIR	Reg. Bank	2000	1742	85	154	19
	PC Reg.	1000	663	93	235	9
	Other Regs	600	571	0	27	2
Kalman	Reg. Bank	2000	1540	185	271	4
	PC Reg.	1000	591	62	346	1
	Other Regs	600	593	0	7	0
TOTAL		10800	8853	488	1408	51

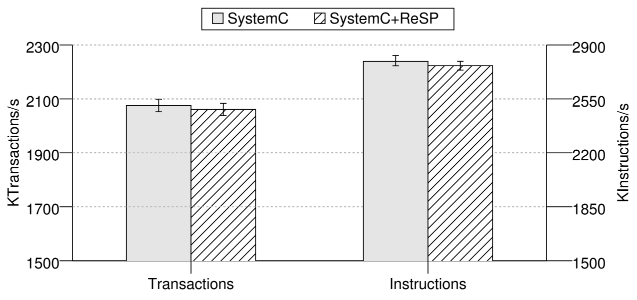

Figure 3: Execution Speed of a both of generic hardware architecture (on the left) and a processor/bus/memory architecture (on the right) measured first using plain SystemC and then using ReSP.

not necessary and, therefore, it is possible to perform fault analysis (a) in a transparent way and (b) significantly reducing the set-up time necessary to be able to carry out the experiments. Only a few works have exploited reflective programming [9], devising, however, solutions strictly related to reliability assessment. In our case, the adoption of SystemC and TLM allows us to propose a flexible framework which is quite innovative also w.r.t. fault injection scenarios.

The possibility of modifying the internal state of the components allows the adoption of a generic functional fault model as well as a radiation induced fault model, such as Single/Multiple Event Upsets. In this work the considered fault model is the soft error or Single Event Effect that represents a transient misbehavior mainly caused by radiations; this behavior affects the devices causing a bit-flip of a value stored in a memory cell. The classes of faults that can be simulated strictly depend on the abstraction level adopted by the component description. At present the components available in ReSP are described at a functional level, hence, the injected faults can only be modeled at behavioral level, rather than structural one; a common approach when considering the complexity of the described cores. At the same time, if a structural description of a component were available, a low abstraction level fault (such as a stuck-at) might be modeled and dealt with.

Moreover, ReSP provides the possibility to automatically instantiate the golden model, i. e., a copy of the system under test used for comparing the faulty system behavior with a fault-free one.

Our fault injection environment can use both the execution modes described in Section 2; during the interactive experimental session, simulation is executed step-by-step; the user can run the simulation and manually suspend it in order to inject faults in the desired storage location. Then, she/he can resume the simulation to monitor the internal state of the architecture under test and to compare it with the golden model (i.e., a fault-free copy of the system under test) in order to analyze the failure evolution. On the other hand, automatic execution can be employed to realize automated fault injection campaigns. The list of faults to be injected is specified through an XML file: each fault is identified in terms of the component and the variable to be changed, the mask to be applied for changing the variable value and the clock cycle at which injection has to be performed. When the experimental session is launched, simulations, one for each element in the list, are performed. The final report, stored in a file, shows for each simulation if the fault has been activated and if it has been detected by fault tolerance features of the circuit under test.

We have used ReSP for reproducing the experimental session proposed in [13]. The purpose of that case study is to evaluate the capabilities of software redundant techniques in detecting faults affecting microprocessors: the system under test is a Leon2 processor running three different applications hardened with software redundant techniques; the initial fault injection environment [4] consisted of an FPGA board emulating the instrumented model of the processor. The same fault injection campaign was repeated by using ReSP: we specified a simple architecture composed of a Leon2 functional model connected to a memory through a bus and we ran the same software applications used in the previous experiment. Several processor registers (e.g.: the Program Counter register, the register bank, the Y register and the PSR register) were indicated as possible fault locations. Table 1 presents the results of our experimental session.

Our approach shows several advantages with respect to the work described in [13]: the capability of performing fault injection by means of introspection allows to carry out experiments in a faster and transparent way (i.e. no modifications to the processor code are needed). It is worth noting that setting up the experimental environment and executing the whole fault injection campaign took only one hour, while instrumenting the processor description for the experiment proposed in [13] took several days. Moreover, our approach does not require complex devices such as FPGAs. Finally, we can perform fault injection experiments at several abstraction levels simply by changing the abstraction level of the components plugged into ReSP.

4. CASE STUDY: DEBUGGING ECOS

The Embedded Configurable Operating System (eCos) is a Real Time Operating System especially designed for high configurability. eCos also features the possibility of running on Symmetric Multi-Processor (SMP) architectures. Unfortunately, in order to enable this last features for the ARM processor, modifications were necessary to the OS source code.

When we first executed a simple application (linked against the modified eCos kernel) composed of two threads writing to serial port, we experimented the fact that only the `printf` instructions contained in one thread were executed. We applied our debugging tools on the programs and we discovered the bug following these steps:

1. A breakpoint was put in the second thread just before the `printf` that was not executed; when the system stopped we issued a step command and verified that actually the execution flow entered the printf routine. When execution ended we examined the *call graph* and discovered that after a certain simulation time the Deferred Interrupt Service routines (DSR) were not executed any more. Thanks again to the call graph we determined the last time the DSR were entered.

2. Using the checkpointing feature we moved the simulation time to the interrupt just after the last successful call to the DSR; by restarting simulation from here we noticed that execution did not enter in the DSR because a global mutex was already locked.

3. By performing a query on the *trace database* we moved simulation back to the last time the lock operation was executed on the mutex and by examining the corresponding code we discovered there was an additional, erroneous call to such lock method.

Using our debugging mechanisms we were able to discover and correct the errors in a much shorter time than what it would have been if we had used a traditional debugging strategy; the main difficulties in debugging such a program came from the fact that the problem was manifesting itself during asynchronous events (i.e. the interrupts). The use of the call graph and of the event database consistently simplified the debugging process.

5. CONCLUDING REMARKS

In this paper we presented ReSP, a hardware simulation platform targeted to Multi-Processor Systems-On-Chip; the platform is based on the integration of Python and SystemC allowing effortless integration of external IPs and custom components. Python augments ReSP with reflective capabilities enabling a fine grained control over simulation and over the internal status of the component modules; this offers advantages, with respect to traditional simulators, in the tasks of reliability analysis, design space exploration and debug and test of the hardware/software system under analysis. Results show that integration among Python and SystemC does not introduce significant overhead over plain SystemC and C++ execution. The effectiveness of our approach was presented through a case study on the software reliability in presence of hardware failures, and a framework for debugging of software and of SystemC hardware models in a virtual platform co-design environment.

6. REFERENCES

[1] C++/python interfacing: pyplusplus. `http://www.language-binding.net`.

[2] Open SystemC Initiative: `http://www.systemc.org`.

[3] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.*, 52(9):1115–1133, 2003.

[4] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. An FPGA-Based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing: Theory and Applications*, 18(3):261–271, 2002.

[5] eCos Operating System. http://ecos.sourceware.org/.

[6] F. Z. et al. Terrestrial cosmic rays and soft errors. *IBM Journal of Research and Development*, 40(1), 1996.

[7] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 327–336, 1989.

[8] M. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[9] E. Martins, C. M. F. Rubira, and N. G. M. Leme. Jaca: A reflective fault injection tool based on patterns. In *Proc. on Dependable Systems and Networks*, pages 483–482, 2002.

[10] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[11] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System–Level Exploration Platform for Network Processors. *IEEE Design and Test of Computers*, pages 2–11, November–December 2002.

[12] A. Pouponnot. A giga instruction architecture (GINA) for the future ESA microprocessor based on the LEON3 IP core. In *DASIA'06*, 2006.

[13] M. Rebaudengo, L. Sterpone, M. Violante, C. Bolchini, A. Miele, and D. Sciuto. Combined software and hardware techniques for the design of reliable ip processors. In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, DFT*, pages 265–273, 2006.

[14] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-Based Architecture Description Language. *sbac-pad*, 00:66–73, 2004.