ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration

Giovanni Beltrame, *Member, IEEE*, Luca Fossati, and Donatella Sciuto, *Senior Member, IEEE*

***Abstract*—This paper presents Reflective Simulation Platform (ReSP), a transaction-level multiprocessor simulation platform based on the integration of SystemC and Python. ReSP exploits the concept of reflection, enabling the integration of SystemC components without source-code modifications and providing full observability of their internal state. ReSP offers fine-grained simulation control and supports the evaluation of different hardware/software configurations of a given application, enabling complete design space exploration. ReSP allows the evaluation of real-time applications on high-level hardware models since it provides the transparent emulation of POSIX-compliant Real-Time Operating Systems (RTOS) primitives. A number of experiments have been performed to validate ReSP and its capabilities, using a set of single- and multithreaded benchmarks, with both POSIX Threads (PThreads) and OpenMP programming styles. These experiments confirm that reflection introduces negligible ($<1\%$) overhead when comparing ReSP to plain SystemC simulation. The results also show that ReSP can be successfully used to analyze and explore concurrent and reconfigurable applications even at very early development stages. In fact, the average error introduced by ReSP's RTOS emulation is below $6.6 \pm 5\%$ w.r.t. the same RTOS running on an instruction set simulator, while simulation speed increases by a factor of ten. Owing to the integration with a scripted language, simulation management is simplified, and experimental setup effort is considerably reduced.**

***Index Terms*—Design exploration, multiprocessor, Python, SystemC, system-level design, system-on-a-chip.**

I. Introduction

MULTIPROCESSOR systems-on-a-chip (MPSoCs) are becoming the prevalent design style to achieve tight time-to-market design goals for high-performance devices, simplifying the verification process and providing flexibility and programmability for postfabrication reuse.

As larger and larger portions of current applications are being implemented in software, their development dominates the cost of the whole system and becomes critical for the development schedule. These issues were already present in "classical" embedded systems, but they have been amplified in MPSoCs which, with explicitly concurrent computation and possibly *real-time* (RT) characteristics, are more difficult to model and analyze. Taking into account hardware and software at the same time is an aid for the design of such systems, allowing software to be developed before the final hardware is ready. In this context, three main methodologies have been devised [1]: 1) the use of a field-programmable gate array (FPGA) emulator; 2) the use of a workstation with appropriate scaffolding; and 3) running the software on a virtual platform (full system simulator). As FPGA prototypes are, in general, not available during the early phases of the design, the use of virtual platforms is becoming widespread.

In order to develop effective virtual platforms of MPSoCs, it is necessary to provide mechanisms for the identification of the complex relationship between application, operating system (OS), and underlying hardware configuration. RT systems also require the accurate modeling of scheduling choices, task interactions, and interrupt response times.

In this paper, we present *Re*flective *S*imulation *P*latform (ReSP), a virtual platform targeted to MPSoC systems. We focus on a component-based design methodology [2], providing primitives to compose complex architectures from basic elements built on top of the SystemC and transaction level modeling (TLM) libraries [3]. The framework exploits Python's *reflective* and *scripting* capabilities, augmenting SystemC components with full observability of their internal structure. This enables runtime composition and dynamic management of the architecture under analysis. The potential offered by the integration of Python and SystemC is used to query, examine, and modify the internal state of hardware models. Such concepts were presented in [4] and, later, extended to include modeling of parallel applications [5] and Real-Time Operating Systems (RTOS) [6]. This paper unifies the approaches, demonstrating how those mechanisms can be used to create a virtual platform to be employed in most of the tasks of system-level design. ReSP has also been extended with capabilities and tools for the modeling of reconfigurable systems and for power analysis and estimation at a high level of abstraction, based on the concepts presented in [7].

Overall, ReSP provides a cosimulation environment that is suitable for RT MPSoCs. This environment is based on a mechanism for the *transparent emulation* of RTOS primitives for any POSIX-compliant application. The most important contributions of the presented simulation platform are the following:

1) *low-overhead reflective support for C++*, augmenting the SystemC library with introspective and reflective

Manuscript received February 24, 2009; revised May 28, 2009, July 14, 2009, and August 6, 2009. Current version published November 18, 2009. This paper was recommended by Associate Editor G. E. Martin.

G. Beltrame is with the European Space Agency, 2200 AG Noordwijk, The Netherlands (e-mail: fossati@elet.polimi.it).

L. Fossati and D. Sciuto are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2030268

capabilities and adding flexibility to the virtual platform technology;
2) *integration with Python*, providing an intuitive simulation management owing to its scripting capabilities;
3) *innovative function call emulation* mechanism, enabling RTOS emulation, preliminary hardware/software codesign, and efficient modeling of reconfigurable architectures;
4) *integration* of the two cited mechanisms into a single simulation platform, providing a usable and flexible environment.

The rest of this paper is organized as follows. Section II presents the most prominent works in the field of system-level simulation and codesign. ReSP's structure and its peculiar features are detailed in Section III. The proposed RTOS emulation system and its extension to support dynamically reconfigurable logic are described, respectively, in Sections IV and V-C, while experimental results showing ReSP's capabilities are contained in Section V. Finally, concluding remarks and directions on future development are presented in Section VI.

II. Related Work

A. Simulation Platforms

Simulation is the most commonly adopted technique for the analysis of functional and nonfunctional requirements of multiprocessor platforms and of embedded systems in general.

CoWare Platform Architect [8] uses SystemC to model and simulate the platform; it also integrates the Processor Designer tool for the specification of instruction set simulators (ISSs). While CoWare Platform Architect contains a wide library of components, integration of additional ones requires a considerable effort.

StepNP [9] is a flexible platform for the simulation and exploration of MPSoCs. Although powerful, it presents several limitations: It features a reduced set of ISSs, and the Open SystemC Initiative (OSCI) TLM standard library is not used for communication, hence reducing interoperability with externally developed components. To augment SystemC with reflective capabilities, SystemC Interface Definition Language, a CORBA-like interface definition language, was created, and each component has to be *manually* modified to be compliant with it. Beltrame *et al.* [7] extended StepNP by introducing the concept of *introspection* to support dynamic switching of simulation accuracy. Even if the approach is innovative, it is considerably intrusive and does not allow complete control over the component models and their internal state. In our platform, the need for ad-hoc wrappers and for IP modifications has been eliminated, owing to the automatic integration between C++ and Python and, consequently, the introduction of reflective properties.

A partially unintrusive approach was adopted by Albertini *et al.* [10], who used the Reflex-SEAL library to add reflective capabilities to SystemC. These capabilities are exploited by a special *white box* component: At every simulator's cycle, the white box inspects the platform's modules and their status, easing the debugging activity. This work does not fully exploit the power of reflection in that it is only used for debugging activities, and by monitoring the component's status at every cycle, simulation speed is consistently decreased.

B. High-Level Modeling of RTOS

Most of the just described platforms can be and are used for the concurrent design [11] of the hardware and software portions of a system. The software part is usually executed on top of an ISS; their low simulation speed pushes for the addition of RTOS models in system-level Hardware Description Languages (HDLs) (as SystemC or SpecC).

The works described in [12]–[15] model the application, the hardware, and the services of the RTOS using the same HDL. Due to the limitations of the typical HDL processing model, application threads are executed sequentially between synchronization points. As such, a tradeoff has to be determined between simulation speed and accuracy of the task interaction models: the more synchronization points, the more the model reflects the actual application behavior and the lower the simulation speed. When the design is refined, the RTOS model can be automatically translated into a standard OS or into custom software services as performed in [16]. In most cases, the use of a standard RTOS is preferred over automatically synthesized services, and in this case, the results obtained using the RTOS model might not be applicable. These methodologies also suffer from the *code equivalence* [1] problem in that the code executed by the model is different from the code executed by the real hardware.

A different technique is employed in [1] and [17], where the authors partially reuse unmodified OS primitives, thus mitigating the code equivalence problem. These OS primitives are ported to the simulator host and are executed natively, guaranteeing high simulation speed, but their timing is not accurately replicated.

Our approach is based on code execution on ISSs, trading-off the minimization of the code equivalence problem with lower simulation speed. In order to enable modeling and analysis of the OS, we implemented a technique known as *system-call emulation*. It is common to find emulation features in ISSs such as *Sim-It ARM* [18], *SWARM* [19], and ArchC [20]. The main drawback of these systems consists in the tight coupling between the compiler and the ISS, thus limiting the portability of the solution: Custom compilers are necessary to enable system-call emulation with such ISSs. With respect to these and other similar systems, our implementation shows the following advantages.

1) *Independence from the cross-compiler toolchain*: Emulated routines are identified with their names; such names are the same no matter what compiler is used. There is no need, for example, to modify the ISS to adhere to the convention with which the software is built [18], [19] or to modify the application code to reflect the internal ISS mechanisms [20].
2) *High interoperability with different ISS types*: An *Interface* is the only component that needs customization to add new ISS models to the simulator.

3) *Extensibility*: As shown in the following, the presented mechanism enables preliminary hardware/software partitioning and codesign. Moreover, a multiprocessor multithreading concurrence manager was implemented, enabling emulation of PThreads.
4) *High code equivalence*: Since only the low-level system calls (SCs) (e.g., `sbrk`) are emulated and the rest of the OS code (e.g., `malloc`) is executed unmodified in the ISS, our method maintains *high code equivalence* with the final software.

C. C++ and Reflection

ReSP's most innovative feature consists of the integration of Python, C++, and SystemC, augmenting C++ with reflective, introspective, and scripting capabilities.

Some programming languages (Java, C#, Python, etc.) implement reflection natively, but C++ supports only a limited set of features under the name of runtime type information, and the metadata required to implement it have to be gathered in a separate step and with separate tools. For example, the Reflex-SEAL library [21] uses GCCXML to parse the source-code header files and to generate an object equivalent structure in XML. This information is then grouped and compiled into a library, which can be linked with the user code to enable reflection. Similar to our approach, SEAL is nonintrusive in the source code, even though it requires user intervention to expose `private` class members.

The idea of augmenting HDLs with the concept of reflection has already been explored in previous works. Doucet *et al.* [22] first analyze the possibility of introducing reflection in system-level language frameworks. They implemented the BALBOA [23] framework, which uses a custom interface description language [Balboa Interface Definition Language (BIDL)] to describe IP structural information (as in [7]) and an interpreted language based on Object Tcl to interpret BIDL data and to assemble the components into a design architecture.

Follow-ups to this approach are presented in [10] (discussed earlier) and [24]. In the latter, introspection and reflection are used to provide enhanced debugging capabilities through automatic test generation and extraction of runtime information from SystemC models. While the overall idea is similar to ReSP's, their approach is complex, requiring the use of Doxygen, of an XML parser, and of a custom abstract system-level description language for extracting reflection data from SystemC code and for representing it. Moreover, their infrastructure is based on the use of the CORBA communication infrastructure, causing a simulation slowdown between 45 and 65 times with respect to unmodified SystemC, which is unacceptable when simulating complex MPSoC designs.

Compared to previous approaches, this work presents the following advantages:

1) reflection of all C++ data structures without requiring access to the source code, simplifying the integration of components and protecting IP rights;
2) fully automated process;
3) scripted language for managing the platform, controlling simulation, and even enabling the description of SystemC components using Python.

In particular, over the other presented approaches, the integration with Python provides scripting capabilities to the framework, which are used to control the computationally intensive parts of the system as often performed in scientific computing [25].

A similar goal could be obtained by attaching the GNU/GDB debugger [26] to the running SystemC thread and using GDB to obtain introspective and some reflective capabilities. With respect to our work, this solution has the following properties: 1) It has lower execution speed; 2) it requires compilation of the IP with special $(-\mathrm{g})$ compilation flags; 3) it does not feature as powerful and complete reflective capabilities; and 4) it does not allow an as intuitive and flexible simulation management as that provided by the integration with Python.

The integration of SystemC and Python is one of ReSP's peculiar features, and even if other works have explored this integration, it was done differently or pursuing different goals. For example, Vennin *et al.* [27] use Python to embed scripting into SystemC modules to reduce the number of lines of code (LOC) used to express a given functionality. This requires the modification of the OSCI SystemC kernel and differs from our goal, which is not the embedding of scripts in SystemC but, vice versa, the use of Python to control SystemC. At the best of the authors' knowledge, ReSP is the only tool providing a complete Python wrapping mechanism for SystemC and TLM designs.

Overall, the techniques presented in this paper overcome many of the limitations of previous approaches, providing a new methodology to create fully observable virtual platforms, using the standard OSCI SystemC kernel and without requiring any modification of existing SystemC IPs. We also introduce the following: 1) an RTOS emulation layer for the evaluation of concurrent applications and of multiprocessor RTOS primitives and 2) an hardware/software partitioning exploration methodology that allows to quickly move any functionality from hardware to software.

III. PROPOSED SIMULATION FRAMEWORK

The goal of ReSP is to give the designer an easy way to generate a high-level model of a system, simulate it in any given configuration, and perform automatic analysis, such as design space exploration (DSE), software optimization, or reliability assessment. To achieve this goal, ReSP starts from SystemC and the OSCI transaction-level modeling library [28] and provides a nonintrusive framework to manipulate SystemC and TLM objects. In the following, the term *framework* indicates the overall ReSP architecture, while *component* describes any top-level SystemC module included into the framework's database (e.g., a processor, a cache, etc.).

The proposed framework is based on the concept of reflection that allows ReSP to view and modify every C++ or SystemC element (variable, method, signal, etc.) specified in any component.

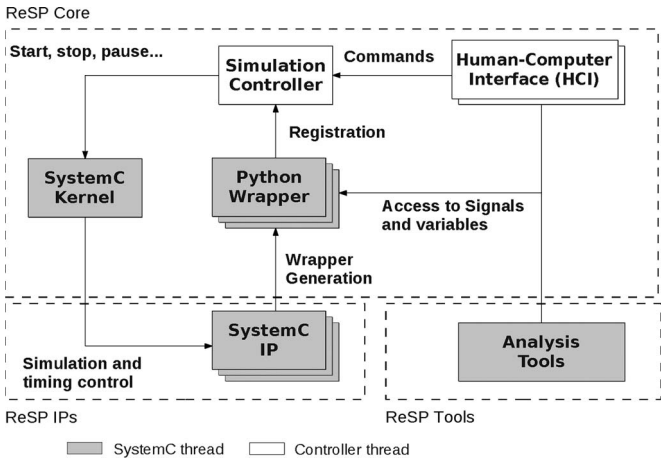

Fig. 1. ReSP system architecture.

In ReSP, SystemC code is directly parsed, and appropriate interface files providing reflection (here called *wrappers*) are automatically generated, integrating standard SystemC TLM IPs with minimum effort.

Fig. 1 shows ReSP structural elements: ReSP *Core*, *IPs*, and *Tools*, described in the following.

A. ReSP Core

ReSP relies on the OSCI *standard* SystemC kernel, as opposed to other works [27] that require special or patched kernels. A Python *wrapper* for the SystemC kernel is automatically created during ReSP's compilation process, adding reflection support and the ability to execute arbitrary function calls to SystemC code from Python. The *simulation controller* is a set of Python classes that translate commands coming from the user into SystemC function calls: It provides functions for running, stepping, pausing, and stopping the simulation.

A novelty introduced by ReSP lies in the Python wrapper generation for SystemC and TLM components. In previous works [7], [8], [10], [27], [29], the developer had to write special interface files or use specific classes in order to add components to the framework's database. Moreover, only the components' characteristics described in those interface files could be used by the simulator. ReSP performs this step automatically,[1] generating Python wrappers by parsing each component's C++ header file. The generation flow is shown in Fig. 2. Each header file is parsed using GCCXML [30], a tool that provides an XML description of the code's abstract syntax tree, i.e., it extracts the name of the declared classes, the signature of their methods, the attributes, etc. The resulting XML description is manipulated by the PyGccxml tool to select all the parts that need to be visible and usable from Python (referred to as *exported* elements), and then the open-source tool py++ [31] generates Python wrapping code. Note that the wrapper creation flow starts from the IP's header files as distributed by the IP provider. Such header files usually contain the whole component's structure; unfortunately, there might be situations

[1]A limited user interaction is needed for corner situations, for example, in case that it is necessary to access from Python a function returning a pointer.

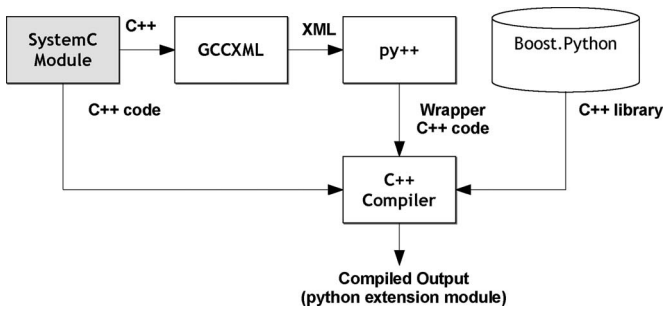

Fig. 2. ReSP wrapper generation flow.

where only the external IP interfaces are distributed: Then, only the available data can be made available from Python.

The SystemC kernel is run in one execution thread, with the rest of the Python wrappers. Human-computer interface (HCI) and tools are executed separately and synchronized with the SystemC kernel when needed (depending on the tool). This way, the user has full asynchronous control of the simulation (the status of the components can be queried and/or modified even while the simulation is running), without loss of consistency. The simulation controller acts as a bridge between the two threads.

A virtual platform can be built using the commands exported by the HCI; components are seen as normal Python classes which are instantiated and connected together with standard Python directives.

B. ReSP Tools

The introduction of reflection paves the way for the development of tools to perform system-level analysis. Any operation that requires observability on both the components' external interface and internal state can be performed through Python wrappers. For example, it is possible to include advanced network traffic analysis (latency, throughput, etc.) by observing the network traffic or to add power modeling to the system by extracting switching activities from the system at runtime (with a technique that is similar to the one described in [32]).

1) Reliability 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 [4]. We have followed the SoftWare-Implemented Hardware Fault Injection [33] approach, based on the modification of the components' internal state and on the simulation of the system behavior in the presence of hardware failures. Works proposed in literature usually pursue fault injection by means of code instrumentation for accessing the internal state of the architecture [34]. When reflection is available, instrumentation is not necessary, and it is possible to perform fault analysis transparently, significantly reducing the experiments' setup time.

We have used ReSP for reproducing the experimental session proposed in [35]. The purpose of that case study is to evaluate the capabilities of software redundant techniques in detecting faults affecting microprocessors; the initial fault injection environment consisted of an FPGA board emulating the

instrumented model of the processor. During the fault injection campaign, several processor registers were indicated as possible fault locations. With respect to related work, the capability of performing fault injection by means of introspection allows one 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 1 h, while instrumenting the processor description for the experiment proposed in [35] 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.

2) Power Modeling: The power modeling framework presented in [32] has been ported to ReSP. This framework allows the generic mapping of power models to TLM *entities* (modules, channels, etc.) at different levels of abstraction. The use of reflection and of the Python scripting language made the implementation noticeably simpler than that in [32], with the integration of the models reduced from the average 192 LOC reported in [32] to an average 50 LOC.

3) Other Tools: Aside from the fault injector, the power modeling framework, and the tools described in Sections IV and V-C, ReSP includes facilities for debugging (in terms both of the IP models and of the software being simulated on top of the hardware platform) and for software profiling. The design and implementation of such tools have been consistently simplified by the ReSP's reflective capabilities.

1) *IP debugger* is implemented through ReSP's Python console: At runtime, simulation can be paused, and owing to reflection, all the members of each C++ class (i.e., the registers, signals, and ports of the IP models) can be listed, examined, and changed if needed. It is also possible to define *callbacks*, methods which are called when user-defined conditions are satisfied (e.g., callbacks which pause simulation when the program counter reached a particular address).
2) *Software debugger* consists of a GDB server, communicating with the GDB debugger through sockets; access to the underlying ISS status is provided by the *processor interface (IF)*, described in detail in Section IV-A.
3) *Software profiler* runs on top of the ISS communicating with it through the *IF*; it enables function profiling, call graph generation, and gathering statistics on the single assembly instructions. Multiprocessor multithreaded programs are supported.

IV. System-Call Emulation

ReSP's characteristics have been exploited to include a system-call emulation subsystem. This is used for the preliminary exploration of concurrent and RT applications, to guide the designer in the choice of the target OS, and as a support for early hardware/software partitioning. ReSP, being a virtual platform, is able to simulate the complete binary, including the RTOS, which will be deployed in the final system, but on top of this, ReSP allows RTOS emulation by redirecting SCs (or any other function call) to the simulator and the OS hosting the simulator.

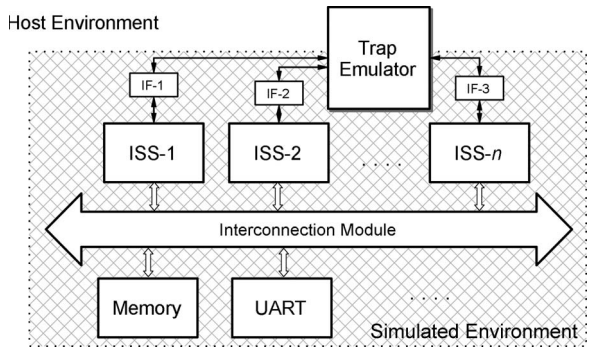

Fig. 3. Organization of the simulated environment, including the system-call emulation module.

System-call emulation is a technique enabling the execution of application programs on an ISS without simulating a complete OS. The low-level calls made by the application to the OS routines (SCs) are identified and intercepted by the ISS and then redirected to the host environment which takes care of their actual execution. Suppose, for example, that an application program contains a call to the `_exit` routine to terminate the application execution. Such a call is identified by the ISS using the mechanisms described hereinafter and routed to the simulator which terminates the simulation, also retrieving the parameter of the `_exit` from the simulated environment in order to determine whether the computation was correctly executed or not.

Having a simulation framework with system-call emulation allows software development to start as early as possible, even before a definite choice about the target OS is performed. An overview of our system-call emulation mechanism is shown in Fig. 3: Each ISS communicates with one centralized trap emulator (TE), the component responsible for forwarding SCs from the simulated environment to the host one. In order to ensure independence between the ISS and the TE, interfaces (IF-1, IF-2, etc.) are created, and communication between the TE and the ISS exclusively takes place through them. With respect to other works, we use the name (i.e., the *symbol*) of the SC routine to identify it: When an application program is loaded for execution, the names of the low-level SCs (e.g., `sbrk`, `_open`, etc.) are associated with their addresses in the application's binary file. During simulation, the ISS checks for those addresses, and when one matches the current program counter, the corresponding functionality of the TE is called and the SC is emulated.

To guarantee the timing accuracy of each I/O-related SC (such as the `write` operation), which would generate traffic on the communication medium, we assume that the SC is executed inside the processor, modeling only the data transfer from processor to memory and vice versa. While this is only an approximation of a real system, accuracy is not severely affected as shown by our experiments.

The remainder of this section is devoted to detail all the components of the emulation mechanism and how it can be used both for hardware/software codesign and for the emulation

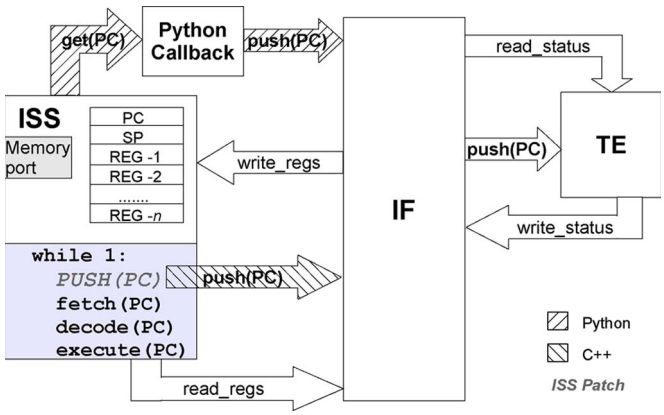

Fig. 4. Communication mechanisms between the ISS and the IF; the C++ or Python parts are alternative solutions.

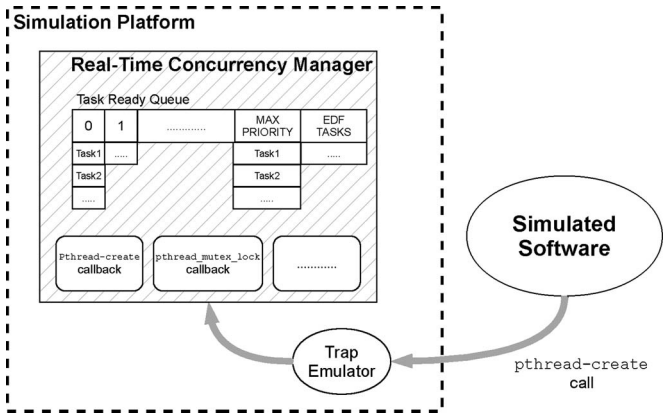

Fig. 5. Detailed structure of the RT concurrence manager.

and management of concurrence in a single- or multiprocessor RT environment.

A. Processor Interface (IF)

Communication between the *Emulator* and the *ISS* is a critical point in the TE design: On one hand, ISSs have to be easily plugged into the system; on the other hand, this communication has to be as fast as possible to guarantee high simulation speed. These requirements are in contrast with each other, and the right tradeoff has to be determined. As shown in Fig. 4, two solutions were identified: One is purely based on the C++ language and does not exploit the *reflective* features of ReSP, and the other uses Python to unintrusively access the ISS internal variables.

Writing pure C++ code sacrifices ease of use and flexibility in favor of simulation speed, meaning that the ISS code has to be slightly modified for the integration with ReSP in order to pass, at every cycle, the program counter to the IF which, in turn, checks if it corresponds to the address of one of the emulated routines. The biggest drawback of this technique is that it requires access to the source code of the ISS, which might not be always available. As an alternative, reflection and integration with Python can be exploited to access all the ISS internal variables, thus unintrusively monitoring the PC, the other registers, and, in general, the whole processor state.

With respect to the communication from the TE to the ISS, both solutions use the same mechanism: The IF exports a set of methods which enable the following: 1) access to the registers (PC, SP, etc.); 2) access to the memory as shown by the processor; and 3) management of the application binary interface of the target architecture. It is worth noting that this interface must have the knowledge of how the ISS is implemented in order to work correctly. This means that specific IFs must be created for the integration of each ISS model.

B. Real-Time Concurrency Manager

The TE was extended for the emulation of concurrence management routines with an additional unit, called *concurrency manager* (CM): In this case, the TE intercepts calls for thread creation, destruction, synchronization, etc. For this purpose, we created a placeholder library containing all the symbols (i.e., the function identifiers) of the PThreads standard, with no implementation present. This ensures that programs using the `pthread` library can correctly compile. During execution, all calls to `pthread` routines are trapped and forwarded to the CM. If the application software is compiled with a recent GNU GCC compiler (at least version 4.2), it is also possible to successfully emulate OpenMP directives. The CM is able to manage shared memory platforms with an arbitrary number of symmetric processors.

RT systems are taken into account by extending the CM to deal with RT requirements and to correctly keep statistics about missed deadlines, serviced interrupts, etc. (Fig. 5).

1) Context Switch Capabilities: To execute different threads on the same processor, *context switch* capabilities are needed; a processor can switch between two threads either when the current thread is blocked (e.g., for synchronization) or when the time quantum associated with the current thread expires. Switching context consists in saving all the ISS registers and restoring the registers for the next thread, like what would happen when using a nonemulated OS, with the only difference that registers are not saved on the stack in memory but in the simulator's space. Access to the ISS's status is provided by the interface described in Section IV-A.

2) RT Scheduler: It is implemented in three different versions: *FIFO*, *Round-Robin*, and *Earliest Deadline First* (EDF). Each task can be assigned a scheduling policy, and tasks with different policies can coexist in the system. Tasks' policies can be varied at runtime either from the application code or by directly interacting with ReSP through the Python console. The latter mechanism has been implemented to enable flexible task management, allowing an effective exploration of the different scheduling policies and priorities and of the different RTOS configurations (Fig. 6). Since scheduling and, in general, task management operations are performed in the host environment, features such as deadlock and race-condition detection are implemented without altering the system's behavior.

3) Interrupt Management: It is composed of an emulated interrupt generator and interrupt service routine (ISR) manager. The interrupt generator emulates external events and stimulates the execution of the ISRs. This enables the analysis of the system behavior under different realistic environmental conditions. When an interrupt is triggered, the manager takes control,

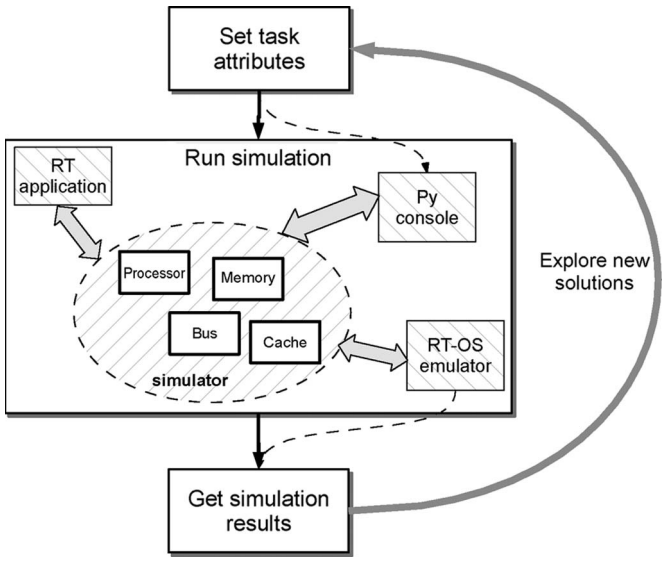

Fig. 6. Exploration of RT policies.

creates a task (each interrupt is mapped to a specified task) for servicing it, and adds the task to the appropriate queue (depending on the task's priority, scheduling policy, etc.); the processor then resumes execution. Interrupts can be triggered in two ways: 1) by an external IP communicating with the processor's interrupt line and 2) by the emulator using an internal timer modeled with SystemC (this features is used to enable the analysis of the system behavior under different realistic environmental conditions).

4) PThreads as an RT Concurrence Model: PThreads are a well-known concurrent application programming interface (API) and are available for most OSs either natively or as a compatibility layer. The PThreads API provides extensions for managing RT threads in the form of two scheduling classes, namely, FIFO and Round-Robin. However, important features such as task scheduling based on deadlines are not present, and this prevents an effective modeling and analysis of a wide range of RT systems. For this reason, our emulation layer extends the PThreads standard with the introduction of the EDF [36] scheduling policy and with the possibility of declaring a task as nonpreemptable. In our implementation of PThread RT extensions, the emulated RT features are compatible with the popular Real-Time Executive for Multiprocessor Systems OSs [37].

5) Python Integration: This feature enables task control from outside the simulated application. It means that the ReSP's interactive shell can manage task priorities, deadlines, etc.; as such, there is no need to modify the simulated software to perform an effective exploration and to analyze the effects of different scheduling policies and/or priorities. Both the system load and RT behavior can be varied during simulation, allowing an efficient exploration of the system's configuration and enabling effective tuning and analysis of RT systems.

C. Extension for Hardware/Software Partitioning Evaluation

While all the features introduced by the TE can be successfully employed for software design and verification, their applicability extends further. The extensions implemented in ReSP allow the designer to "trap" any routine, not just the ones belonging to the OS or to the PThreads library. This enables an effective and fast evaluation of the different possible implementations of a given functionality f.

1) Execution of the *unchanged software routine*: No trap is installed for f, and the functionality of f is simulated on the ISS.
2) *Emulation*: A trap is installed for f, and its behavior is emulated. Since it is possible to associate a custom delay d to the trap handler, we can explore how the timing of the system would be if f were implemented with a hardware module of delay d.
3) Implementation through a *hardware module*: The trap handler triggers the execution of a SystemC defined hardware module containing the functionality of f; this step is used to refine the emulation of 2).

By associating a custom delay d with a trap handler (i.e., with an emulated routine), the described techniques can be used to explore the system behavior, considering different latencies for threading and synchronization primitives, helping in the choice of the OS, and, for example, deciding whether hardware managed threading is needed.

D. Python-Implemented Emulation

As discussed, it is possible to specify a custom behavior and/or latency for each emulated routine. In particular, when a trapped routine is encountered, a user-defined functor (the *trap handler*) is called. This functor is part of the simulator, and as such, it has access to both the simulated and the host environments. For example, in the context of RTOS emulation, there are functors for *thread creation*, *thread destruction*, *memory allocation*, etc. A functor can be defined either in C++ or in Python. The former mechanism requires modifications to the source code in order to add a new functor or modify an existing one; therefore, it is not targeted to performing fast exploration of the different implementation alternatives for a functionality. Despite being slower, as shown in Section V, the latter solution enormously increases the productivity, enabling the scripted implementation of any trap handler.

V. Experimental Results

This section presents the set of experiments that were executed to assess ReSP's performance and capabilities in the different scenarios presented throughout this paper.

A. ReSP Overhead

To ensure that the additional control layers on top of SystemC would not excessively affect simulation speed, a set of plain SystemC modules was connected both via standard C++ code and via ReSP, and the two architectures were simulated, and the execution time was recorded. This experiment does not include the emulation layer introduced in Section IV since its aim is to show that the integration with Python (renown for its

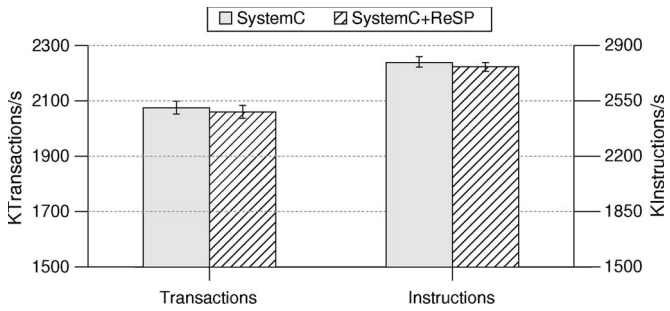

Fig. 7. Execution speed of both (on the left) a generic hardware architecture and (on the right) a processor/bus/memory architecture measured first using plain SystemC and then using ReSP.

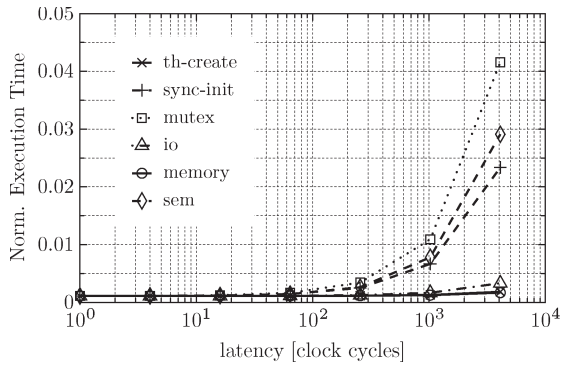

Fig. 8. Average behavior of eight benchmarks considering varying latencies for the different SCs; each point is the average of the execution on 2, 4, 8, and 16 cores/threads.

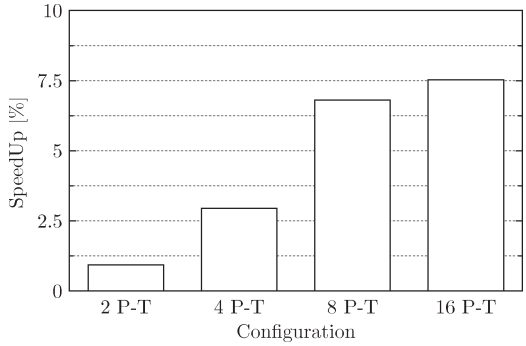

Fig. 9. Speedup obtained by the `lu` benchmark for hardware-implemented synchronization primitives.

slow execution speed) does not add an excessive overhead over plain SystemC simulations.

The left side of Fig. 7 shows the first experiment that was set up to measure the transactional speed of the system, using basic master and slave components exchanging characters. The right side of Fig. 7 shows the results of the execution of a full architecture that was created by connecting a functional LEON2 processor model and the TLM Programmer's View memory and bus. The number of instructions per second, obtained both using native execution and execution inside ReSP, is shown. Results show a small performance penalty (below 1%) due to the additional software layer introduced by Python. All the experiments were hosted on a 2-GHz Intel Core 2 Duo System with 4 GB of RAM running Gentoo Linux.

By integrating with Python, ReSP provides a considerable increase in productivity: An average `main` file for a SystemC multiprocessor platform (including processor instantiation, serial ports, timers, and all the other necessary peripherals) is around 350 LOC, including the code to parse command line options, the inclusion of headers files, explicit memory management, etc. Writing a "clever" `main` file might reduce the number of compilations, but the amount of LOC is bound to increase, and the solution is not general. ReSP provides a general solution with Python scripting, which is more than three times more compact in terms of LOC and development time when compared to C++ [38], [39]. This means that new configurations can be explored and new tools can be developed with a significantly reduced coding effort and without significant penalty in execution time (as shown in the aforementioned experiments). Such a solution is new in the context of HDLs and simulation platforms, but it is well known in scientific computing, where a scripted language is often used to control the computational part of applications written in C++ [25].

B. System-Call Emulation

The trapping and emulation capabilities of the system can be used for the following: 1) hardware/software codesign; 2) the estimation of the parallelism embedded in a given software application; and 3) the estimation of thread management overhead.

All tests have been executed using ReSP on a multi-ARM architecture consisting of a variable number of cores with caches, a shared memory, and timers, interrupt controllers, and serial ports as needed by the OS (for relevant experiments), all interconnected by a shared bus, as shown in Fig. 3. The processors are clocked at 500 MHz, and the bus at 250 MHz.

1) OS Emulation for Codesign: To prove the usefulness of the methodology for codesign, we analyzed the impact of OS latencies on a set of eight parallel benchmarks chosen from the OpenMP Source Code Repository [40]. OS primitives were divided into six classes: thread creation (*th-create*); synchronization initialization, e.g., mutex, semaphore, and condition variable creation (*sync-init*); mutex locking and unlocking (*mutex*); semaphore waiting and posting (*sem*); memory management (*memory*); and general I/O (*io*). Since it is possible to associate custom latencies with emulated routines, we can observe how different values (corresponding to different system configurations) affect the system. Fig. 8 shows the average behavior of the eight benchmarks when the number of cores, as well as the number of OpenMP threads, ranges from 2 to 16 in powers of 2. On each of these hardware configurations, the benchmarks were run with exponentially increasing latencies for each class, yielding a total of 1344 simulations. The trend is that, for increasing latency, synchronization primitives are the ones that affect execution time most, while I/O and memory management have negligible effect.

This kind of analysis can guide the designer's decision concerning the OS architecture and configuration, also taking into account hardware-supported OS mechanisms.

As an example, Fig. 9 shows `lu`'s performance when a hardware accelerator for managing synchronization primitives

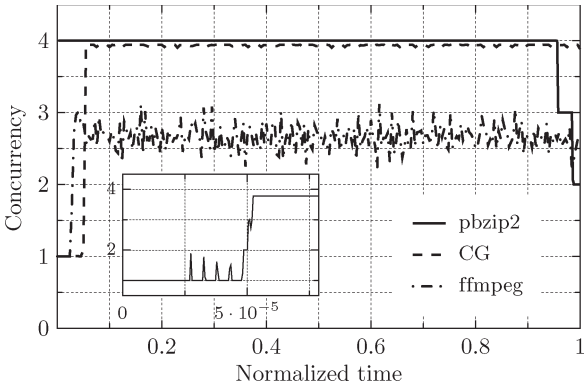

Fig. 10. Concurrency profiles (i.e., number if active processors) on a four-core architecture; the initialization of pbzip2 is shown in detail.

is inserted in the system. Assuming a memory-mapped module with a four-cycle latency, we observe an increasing speedup with the number of processors/threads (P-T), saturating around eight processors (mainly due to the lack of parallelism of the benchmark).

All these experiments were run semiautomatically, without any modification to the baseline SystemC and application code. This is due to the emulator subsystem and to the integration with Python's console, since simulation management is consistently simplified with respect to standard simulation platforms controlled by custom configuration files.

2) Software Overhead Estimation: The proposed methodology can also be used to estimate the inherent parallelism of an application and the overhead introduced by thread management. We examined two multithreaded open-source applications and an OpenMP benchmark: *ffmpeg*, a powerful audio and video encoder/decoder, *pbzip2*, a parallel block compression algorithm, and *CG*, a scientific application kernel from the NASA Parallel Benchmark (NPB) suite [41]. For these tests, a four-core four-threaded architecture was used.

It is worth noting that such applications run natively on Linux, and their source code is left completely unmodified when executing them on the simulator. In addition, they are linked with the final OS, and only the low-level OS routines are emulated. With respect to related works (Section II-B), this is an advantage since the same exact executable file can be used to simulate the whole software infrastructure (including the OS) or just the application kernel (when OS emulation is enabled).

According to Amdhal's law [42], the main idea of these tests is to run the application setting of all the concurrence-related latencies to zero, hence exposing the application's inherent parallel behavior. The intuition behind this is that, in the ideal case, when thread management costs 0, the speedup of a perfectly parallelized application given by the execution on n threads is equal to n. In practice, this does not happen, and the speedup is $m \leq n$, with some overhead introduced due to the parallelization of the application (which introduces different data structures, algorithms, etc.) and due to the sequential application portions.

Determining the number of active processors for each clock cycle leads to generating the concurrence *profiles* shown in Fig. 10: They show the number of active processors during the application runtime. To measure the overhead introduced by thread management, we compare the average concurrence expressed by each application, with the actual speedup obtained against single-threaded execution as $overhead = (1 - speedup/concurrence) \times 100$.

TABLE I
APPLICATION RUNTIME AND OVERHEADS

Name	Concurrency	Speedup	Overhead
ffmpeg	2.6	2.3	11%
pbzip2	3.4	2.9	15%
CG	3.8	3.5	9%

The concurrence profile of `ffmpeg`, transcoding (decoding and encoding) a short clip from MPEG1 to MPEG4, shows that, due to data dependences and to the structure of the ffmpeg's code, the four processors are never fully exploited, with an average of 2.6 processors working at the same time. When compared to a single-core architecture, ffmpeg is 2.3 times faster, yielding an 11% overhead. `pbzip2`, instead, offers more exploitable concurrence as its algorithm has less data dependences. After a very small initial setup time, pbzip2 fully exploits the available resources. However, even if the average concurrence expressed by pbzip2 is 3.4, its speedup on four cores is only 2.9, resulting in a 15% overhead. Finally, `CG` shows the benefits of OpenMP programming, with a very smooth concurrence profile, locked to 3.9 after the initialization phase and averaging at 3.77. CG's speedup amounts to 3.55, yielding a small 9% overhead introduced by the compiler's OpenMP runtime (GNU libgomp in our case). All results are summarized in Table I.

3) Accuracy of OS Emulation: To evaluate the performance and accuracy of OS emulation with respect to a real OS, 12 OmpSCR benchmarks were run with the RTOS eCos [43], using a four-core platform. A large set of eCos SCs was measured, running six of these benchmarks as a training or calibration set, and the average latency of each class of SCs was determined. The Lilliefors/Van Soest test of normality [44] applied to the residuals of each class shows evidence of nonnormality ($L = 0.30$ and $L_{\text{critical}} = 0.28$ with $\alpha = .01$), but given that the population variability remains limited (with a within-group mean square $MS_{S(A)} = 7602$ clock cycles), it can be assumed that each average latency is representative of its class.

The derived latencies were introduced for each SC in our OS emulation system, and the remaining six benchmarks (used as a validation set) were executed. Since profiling did not include all functions used by the OS, and for which the latency was considered zero, the overall results were uniformly biased for underestimation. This bias can be easily corrected considering the average error, leading to an average error of $6.6 \pm 5.5\%$, as shown in Fig. 11. Even with this simple scheme, the methodology can very well emulate the behavior of a specific OS with minimal error, particularly considering that full code equivalence is present for the application and library functions, but threading, multiprocessor management, and low-level OS functions are emulated.

In addition, the use of the OS emulation layer introduces a noticeable speedup (13.12 ± 6.7 times) when compared to running the OS on each ISS. This is due to several factors,

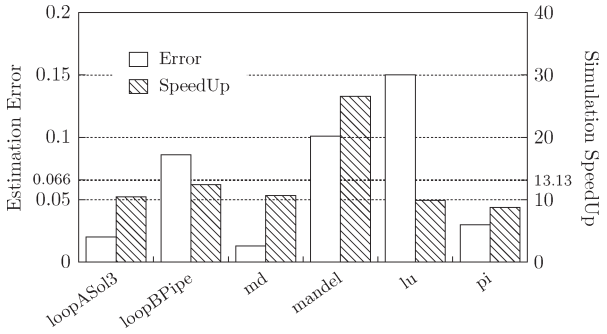

Fig. 11. Simulation speedup and estimation error using the emulation layer instead of eCos.

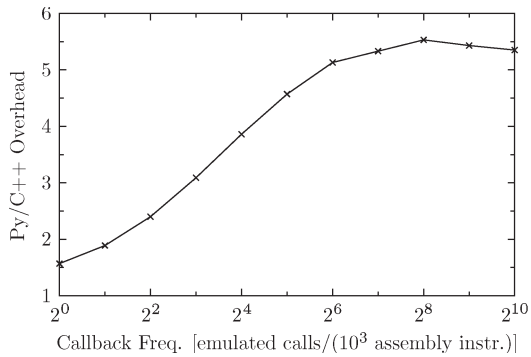

Fig. 12. Speedup of C++ with respect to Python routine emulation while varying the frequency of the calls to the emulated functionality.

including the absence of some hardware components such as debugging serial ports and timers (the TE implements a terminal in the host OS, and the configuration manager uses SystemC and its events to keep track of time), and the fact that, in our mechanism, idle processors do not execute busy loops, but they are, instead, suspended. The latter is implemented by trapping the busy loop wait function in the TE and redirecting it to a SystemC wait() call.

4) Mixing Python and C++: As described in Section IV, the behavior of the emulated routines (which are not limited to the OS routines) can be written either using C++ or Python. In these experiments, the overhead introduced when writing the emulated functions in Python was measured. Fig. 12 shows the overhead of using Python-emulated functions with respect to C++: Python is ∼5.5 times slower. Note how, when the frequency of the emulated calls is low, there is not much difference in using Python or C++, since simulation time is mainly due to the ISS. As the frequency grows, most of the time is spent in routine emulating, and the difference in the two emulation mechanisms is more noticeable.

5) Emulation of RT Systems: As described in Section IV-B, the emulation subsystem can also be used for the analysis of RT applications. In these experiments, we assume that the system is subject to a mixed applications workload: a computationally intensive element with soft RT constraints and a set of elements with very strict hard RT characteristics, here called computational and RT parts, respectively. This model well represents applications such as observation spacecraft payload, where massive data processing is needed with high availability, while requiring response to external stimuli within a given

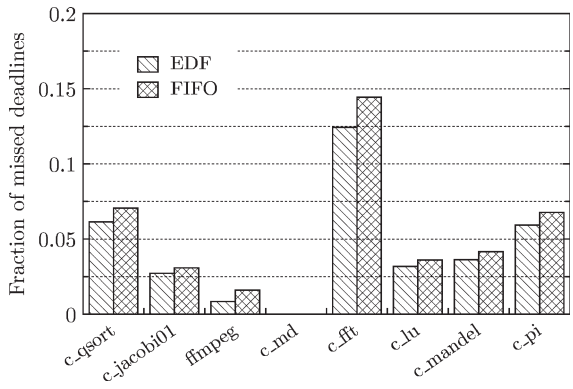

Fig. 13. Fraction of missed deadlines with different schedulers and high RT workload (1 kHz).

time. The number and parameters of both the computational and RT parts vary and strictly depend on the system being considered.

Also in this case, our methodology allows one to run this set of experiments with reduced effort: No modification of the source code is necessary, and the same binaries can be reused for a large set of experiments. Indeed, the SystemC platform files could be modified to produce the same result, but the solution would not be general and not applicable to any platform or application like in the case of our methodology.

Fig. 13 shows how the methodology is used to determine the best scheduler (among the considered ones) for the system. We ran the benchmarks previously presented as computationally intensive applications, while the RT tasks were implemented by synthetic functions, with varying deadlines. These functions can be categorized as follows: 1) housekeeping: scheduled regularly, such as performing sanity checks, repetitive tasks, etc., and 2) response to external events: when an alarm is fired, its response is usually required within a given deadline. We see that the EDF scheduler performs better for all the benchmarks; thus, it is the best choice for the tested system.

C. Case Study: High-Level Modeling of Dynamic Reconfigurability

Aside from the simulation and analysis of standard multiprocessor architectures, ReSP is able to model dynamically reconfigurable systems. We assume that reconfiguration (as it happens in many reconfigurable embedded architectures [45]) is software controlled and that the reconfigurable blocks are available to the system in the form of embedded FPGA blocks, connected to an on-chip interconnection (a bus or network-on-chip). In our system, we speak of *partial internal dynamic* reconfigurability in the sense that reconfiguration takes place *internally* to the system-on-a-chip; *partial* and *dynamic*, respectively, indicate that it is not necessary to reconfigure the whole FPGA and that reconfiguration can take place at runtime, while the rest of the system is operating.

The novelty of this approach resides in the reconfiguration manager (RM) and in the model of the eFPGA modules. On the one hand, the use of the RM is totally transparent to the

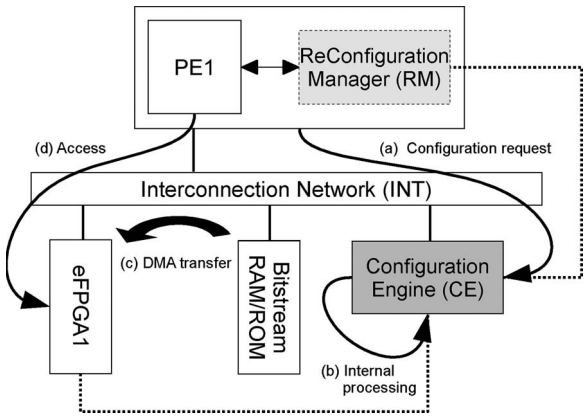

Fig. 14. Behavior and delays modeled by the CE.

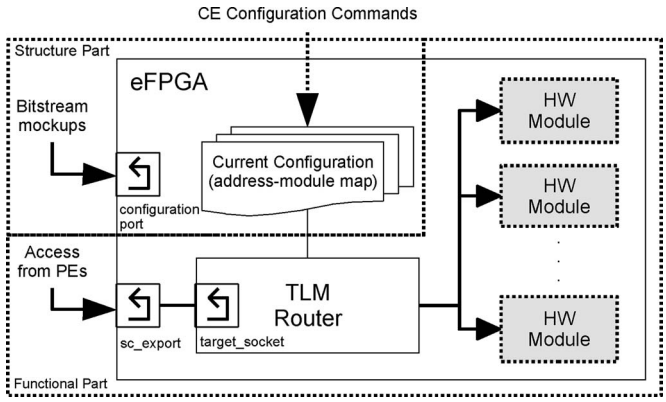

Fig. 15. eFPGA model.

software designer: The RM can be configured to mark functions or method calls as "reconfigurable functions," and whenever the software performs a call to such methods, the call is routed to the properly configured hardware component. On the other hand, the eFPGA models implement this functionality using C++, SystemC, or Python code, using specified area, bitstream size, and delay. This paves the way for a fast and effective exploration of the possible mappings between functionalities (method calls in software) and either their software or hardware implementation.

More in details, our reconfigurability modeling subsystem is composed of the following elements.

1) RM: The RM is a C++ component that exists uniquely in the frame of the simulator, i.e., it has no real hardware counterpart. Its purpose is to *model* the adaptive behavior of a system for DSE purposes, without enforcing any particular implementation of the reconfiguration mechanism, that is instead modeled by the configuration engine (CE). Note that the RM is a modified version of the TE previously presented in the context of RTOS emulation.

Summarizing, the RM is the module which decides whether a given functionality should be executed using dedicated reconfigurable hardware or software. This element communicates directly with the processors [also called processing elements (PEs)] and with the CE in order to trigger the computation of a functionality in hardware and (if necessary) the reconfiguration process. As the RM is initialized, it carries no information about which functions should be marked for hardware implementation. Function registration and deregistration are totally dynamic, and they can even be performed during simulation.

2) CE: The CE represents the hardware component that is responsible for the execution of the actual reconfiguration requests. Considering a transaction-level model of reconfigurability, the CE has to model with reasonable accuracy: 1) the communication delay between the PEs and the CE; 2) the internal processing of the CE that has to keep track of which components have been configured, the available eFPGA blocks, etc.; 3) contention, bandwidth usage, and latency of the transfer of the bitstream from memory to the eFPGA block; and 4) the actual access latency to the block. Fig. 14 shows how this paper addresses these issues. The RM communicates, using the TLM2 standard transaction payload, with the CE through the requesting PE port, covering 1). Once the RM has instructed the CE to provide a certain functionality, the CE reserves, in the eFPGA model, a block of sufficient size (whether by using free space or by erasing other blocks); a user-defined delay can be specified during this operation 2). Then, the CE works as a direct memory access (DMA) engine, transferring a mockup of the bitstream from memory to the selected block, modeling 3).

The sizes and the availability of reconfigurable blocks in the eFPGA model are left for the designer to define. In our experiments, as proof of concept, we consider two medium-sized (100k equivalent gates) blocks for each eFPGA.

3) eFPGA Model: The eFPGA model consists of two parts: structure and functionality. The structure models the internal networking of the FPGA block, its available area, and resources. The functionality consists of a standard TLM2 router that forwards transactions to the appropriate SystemC module or Python function, as shown in Fig. 15. The eFPGA has two TLM ports: a configuration port, used to receive bitstreams, and a data port, used by the PEs to access hardware modules. The structure part contains a configuration data structure that stores information on the currently configured blocks, the references to their implementation, address spaces, etc.

The extensions for modeling dynamically reconfigurable systems have been validated using the NPB [41] suite. The performed tests aim at evaluating the framework for the exploration of different reconfiguration alternatives. To prove this concept, the block tri-diagonal (BT) application of the NPB suite was tested, and a subset of its routines was examined in order to determine the advantages and disadvantages of their implementation using reconfigurable hardware. Concerning reconfigurable logic, we assume the following.

1) There are two possible eFPGA configurations: one module with one reconfigurable block and one module with two blocks.
2) The blocks are medium-sized, of roughly 2000 logic cells, and the configuration bitstream for each block is about 500 kbit. This number is based on Xilinx FPGA bitstreams [46].
3) Whenever a block is reconfigured, its bitstream is downloaded from RAM via DMA on the shared bus, and the

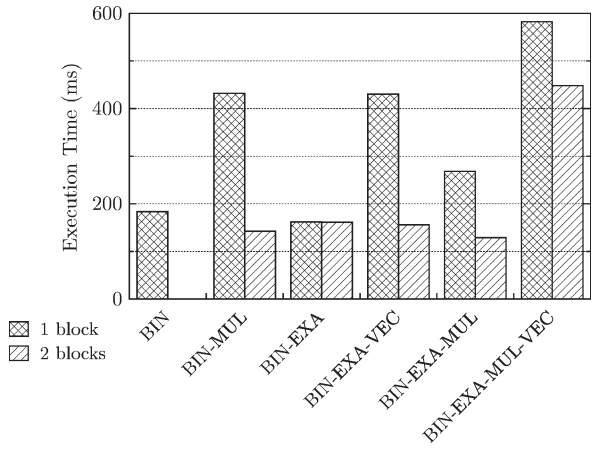

Fig. 16. Benchmark execution time as functions are moved to reconfigurable logic, with one and two blocks available.

block is ready to be used after transfer plus a setup time of 10 μs.

4) The substitution policy is random.

The application was profiled with ReSP, determining the four most computationally intensive routines (here called `BIN`, `MUL`, `VEC`, and `EXA`, all routines for the manipulation of matrices of floating point numbers). Those are ideal candidates for a hardware implementation, as they are self-contained, they operate on a small set of parameters, and they do not have any side effect. We assume that the implementation of these functions fits in the available reconfigurable blocks, and that their latency in clock cycles is equal to the number of floating-point operations executed by each of them. Fig. 16 shows the performance trend as these blocks are turned to hardware; this trend is due to the reconfiguration and setup time of the eFPGA and the fact that BIN, MUL, and VEC, called alternatively by the four processors, compete for the block ownership. By using a two-block eFPGA, the situation improves as BIN and MUL (or BIN and VEC) can be called alternatively without delays, but the combination of BIN, MUL, and VEC still incurs long delays.

When compared to SystemC-only approaches, our solution provides a consistent increase in productivity: Without modifying the source code of an application, it is possible to switch a module from software to hardware by changing a single parameter from ReSP's console. As stated before, this could be implemented directly in SystemC, but the solution would not be general and applicable to any platform or application.

VI. Conclusion

In this paper, we have presented ReSP, a hardware simulation platform targeted to MPSoCs. This platform is based on the integration of Python and SystemC, allowing easy integration of external IPs and custom component models. Python augments SystemC with reflective capabilities and enables a fine-grain control over simulation and full observability of the internal state of the platform components. The main advantage with respect to traditional compiled simulation is that it eases the tasks of reliability analysis, DSE, and debug and test of the hardware/software system under analysis. We have also introduced a powerful RTOS emulation mechanism that allows the following: 1) the early DSE of the RTOS and its configuration; 2) the measurement of the inherent parallelism of concurrent application and their thread management overhead; and 3) the high-level DSE of reconfigurable platforms. The effectiveness of the approach was proven through a series of experiments, analyzing several applications using either PThreads or OpenMP programming models and modeling reconfigurable embedded FPGA components. ReSP does not introduce any significant overhead when compared to plain SystemC, and its OS emulation capabilities guarantee a speedup up to a factor of 13 with respect to executing a real OS on an ISS, with a low 6.6 $\pm$ 5.5% timing estimation error.

References

[1] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya, "Automatic generation including fast timed simulation models of operating systems in multiprocessor SoC communication design," in *Proc. DATE*, Mar. 2002, pp. 620–627.

[2] W. Cescirio, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore SoCs," in *Proc. DAC*, Jun. 2002, pp. 789–794.

[3] Open SystemC Initiative. [Online]. Available: http://www.systemc.org

[4] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "ReSP: A non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration," in *Proc. ASPDAC*, Jan. 2008, pp. 673–678.

[5] G. Beltrame, L. Fossati, and D. Sciuto, "Concurrency emulation and analysis of parallel applications for multi-processor system-on-chip co-design," in *Proc. CODES/ISSS*, Oct. 2008, pp. 7–12.

[6] G. Beltrame, L. Fossati, and D. Sciuto, "A real-time application design methodology for MPSoCs," in *Proc. DATE*, Apr. 2009, pp. 1–6.

[7] G. Beltrame, D. Sciuto, C. Silvano, D. Lyonnard, and C. Pilkington, "Exploiting TLM and object introspection for system-level simulation," in *Proc. DATE*, Mar. 2006, pp. 100–105.

[8] CoWare Platform Architect. [Online]. Available: http://www.coware.com/products/platformarchitect.php

[9] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 7, pp. 667–680, Jul. 2006.

[10] B. Albertini, S. Rigo, G. Araujo, C. Araujo, E. Barros, and W. Azevedo, "A computational reflection mechanism to support platform debugging in SystemC," in *Proc. CODES +ISSS*, Oct. 2007, pp. 81–86.

[11] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, Apr. 2003.

[12] R. L. Moigne, O. Pasquier, and J.-P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," in *Proc. DATE*, Feb. 2004, pp. 82–87.

[13] E. Huck, B. Miramond, and F. Verdier, "A modular SystemC RTOS model for embedded services exploration," in *Proc. DASIP*, 2007.

[14] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS modeling for system level design," in *Proc. DATE*, 2003, pp. 130–135.

[15] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco, "POSIX modeling in SystemC," in *Proc. Asia South Pacifi Des. Autom. Conf.*, Yokohama, Japan, Jan. 2006, pp. 485–490.

[16] P. Destro, F. Fummi, and G. Pravadelli, "A smooth refinement flow for co-designing HW and SW threads," in *Proc. Des. Autom. Test Eur. Conf.*, Apr. 2007, pp. 105–110.

[17] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada, "RTOS-centric hardware/software cosimulator for embedded system design," in *Proc. CODES/ISSS*, Sep. 2004, pp. 158–163.

[18] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *Proc. Des. Autom. Test Eur.*, 2003, pp. 556–561.

[19] Software ARM. [Online]. Available: www.cl.cam.ac.uk/~mwd24/phd/swarm.html

[20] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "ArchC: A SystemC-based architecture description language," in *Proc. 16th SBAC-PAD*, 2004, pp. 66–73.

[21] S. Roiser and P. Mato, "The SEAL C++ reflection system," in *Proc. Comput. High Energy Phys. Nucl. Phys.*, Sep. 2004, pp. 437–446.
[22] F. Doucet, S. Shukla, and R. Gupta, "Introspection in system-level language frameworks: Meta-level vs. integrated," in *Proc. Des. Autom. Test Eur. Conf.*, 2003, pp. 382–387.
[23] F. Doucet, S. Shukla, R. Gupta, and M. Otsuka, "An environment for dynamic component composition for efficient co-design," in *Proc. Des. Autom. Test Eur.*, 2002, pp. 736–741.
[24] H. Patel, D. Mathaikutty, D. Berner, and S. Shukla, "CARH: Service-oriented architecture for validating system-level designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 8, pp. 1458–1474, Aug. 2006.
[25] T.-Y. Yang, G. Furnish, and P. Dubois, "Steering object-oriented scientific computations," in *Proc. TOOLS 23*, Jul.–Aug. 1, 1997, pp. 112–119.
[26] Gdb Debugger. [Online]. Available: http://sourceware.org/gdb/
[27] J. Vennin, S. Penain, L. Charest, S. Meftali, and J. Dekeyser, "Embedded scripting inside SystemC," in *Proc. FDL*, 2005, pp. 373–385.
[28] L. Yu, S. Abdi, and D. Gajski, "Transaction level platform modeling in SystemC for multi-processor designs," Center Embedded Comput. Syst., Univ. California, Irvine, Irvine, CA, 2007.
[29] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A system-level exploration platform for network processors," *IEEE Des. Test Comput.*, vol. 19, no. 6, pp. 17–26, Nov./Dec. 2002.
[30] GCCXML. [Online]. Available: http://www.gccxml.org/
[31] C++/Python Interfacing: Pyplusplus. [Online]. Available: http://www.language-binding.net
[32] G. Beltrame, D. Sciuto, and C. Silvano, "Multi-accuracy power and performance transaction-level modeling," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 10, pp. 1830–1842, Oct. 2007.
[33] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003.
[34] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.
[35] 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. DFT VLSI Syst.*, 2006, pp. 265–273.
[36] M. Kargahi and A. Movaghar, "Non-preemptive earliest-deadline-first scheduling policy: A performance study," in *Proc. 13th IEEE ISMASCTS*, Sep. 2005, pp. 201–210.
[37] RTEMS Home Page. [Online]. Available: http://www.rtems.com/
[38] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000.
[39] J. K. Ousterhout, "Scripting: Higher-level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, Mar. 1998.
[40] A. J. Dorta, C. Rodríguez, and F. de Sande, "The OpenMP source code repository," in *Proc. 13th Euromicro Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2005, pp. 244–250.
[41] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA Ames Res. Center, Moffett Field, CA, Oct. 1999.
[42] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.
[43] Ecos Operating System. [Online]. Available: http://ecos.sourceware.org/
[44] H. Lilliefors, "On the Kolmogorov-Smirnov test for normality with mean and variance unknown," *J. Amer. Stat. Assoc.*, vol. 62, pp. 399–402, 1967.
[45] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002.
[46] FPGA and CPLD Solutions, San Jose, CA: Xilinx, Inc. [Online]. Available: http://www.xilinx.com/

Giovanni Beltrame (M'08) received the M.Sc. degree in electrical engineering and computer science from the University of Illinois, Chicago, in 2001, the Laurea and Ph.D. degrees in computer engineering from the Politecnico di Milano, Milan, Italy, in 2002 and 2006, respectively, and the M.S. degree in information technology from Cefriel, Milan, in 2002.

After being a Research Fellow with the European Space Agency, Noordwijk, The Netherlands, he currently holds a position as a Contractor. His research interests include modeling and design of embedded systems, artificial intelligence, and robotics.

Luca Fossati received the Laurea degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2006 and the M.Sc. degree in electrical engineering and computer science from the University of Illinois, Chicago, in 2007. He is currently working toward the Ph.D. degree at Politecnico di Milano.

His research interests include modeling and design of embedded systems, compilers, and genetic algorithms.

Donatella Sciuto (S'84–M'87–SM'08) received the Ph.D. degree in electrical and computer engineering from the University of Colorado, Boulder, and the MBA degree from the Bocconi University, Milan, Italy.

She is currently a Full Professor in computer science and engineering with Politecnico di Milano, Milan, Italy. Her research interests include embedded systems design methodologies and architectures. She has published more than 200 papers in international journals and conferences.

Dr. Sciuto is a member of International Federation for Information Processing 10.5 and European Design and Automation Association. She has been or is a member of different program committees of Electronic Design Automation (EDA) conferences and has been an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS. She is an Associate Editor of the *Journal Design Automation of Embedded Systems* and of the IEEE EMBEDDED SYSTEMS LETTERS. She has been the Program Chair of DATE 2006, the Vice General Chair of DATE 2007, and the General Chair in 2008. She is the Vice General Chair of ESWEEK 2009 and will be the General Chair in 2010. She is the VP of Finance of the IEEE Council of EDA.