esa DOCUMENT

ESA STUDY CONTRACT REPORT - SPECIMEN

No ESA Study Contract Report will be accepted unless this sheet is inserted at the beginning of each volume of the Report.

ESA Contract No: 20921/07/NL/JD * **ESA CR( )No:**

SUBJECT: Technical on-site support for ReSP (Reflective Simulation Platform)

CONTRACTOR: Politecnico di Milano

CONTRACTOR'S REFERENCE:

No. of Volumes: 1

This is Volume No.: 1

ABSTRACT:
The aim of the contract is to bring ReSP to a state where both HW and SW can be concurrently simulated and explored, providing a proof-of-concept parallel application running on a multi-core virtual platform.
This proposal contains three aspects of innovation: (1) manipulation of already existing software to adapt it to multi-processor application (2) design space exploration of the hardware platform (3) fault-tolerance studies and reliability techniques applied to the newest embedded Multi-Processor systems.

Mr. Fossati stay at ESTEC will involve research and development activities in the Microelectronics Lab, with the application of technologies developed for the commercial market to space applications.

The work described in this report was done under ESA Contract. Responsibility for the contents resides in the author or organisation that prepared it.

Names of authors: Luca Fossati

** **NAME OF ESA STUDY MANAGER**: ** **ESA BUDGET HEADING**:

DIV:

DIRECTORATE:

* Sections to be completed by ESA
** Information to be provided by ESA Study Manager

European Space Agency
Agence spatiale européenne

ESTEC
Keplerlaan 1 - 2201 AZ Noordwijk - The Netherlands
Tel. (31) 71 5656565 - Fax (31) 71 5656040

Technical on-site support for ReSP (Reflective Simulation Platform)

Final report

prepared by/*préparé par*	Luca Fossati
reference/*réference*	ESTEC Contract No. 20921/07/NL/JD
issue/*édition*	1
revision/*révision*	1
date of issue/*date d'édition*	15/05/2008
status/*état*	Final
Document type/*type de document*	Final report of a stage period at ESA ESTEC

A P P R O V A L

Title *titre*	Technical on-site support for ReSP (Reflective Simulation Platform)	issue 1 *issue*	1

author *auteur*	Luca Fossati	date 15/05/2008 *date*

approved by *approuvé by*		date *date*

C H A N G E   L O G

reason for change /*raison du changement*	issue/*issue*	revision/*revision*	date/*date*
First version	1	0	15/05/2008

C H A N G E   R E C O R D

Issue: 1 Revision: 1

reason for change/*raison du changement*	page(s)/*page(s)*	paragraph(s)/*paragraph(s)*

TABLE OF CONTENTS

1 INTRODUCTION....................1

2 BACKGROUND....................4
2.1 Virtual Platforms 4
2.2 Transaction Level Modelling....................5

3 OVERALL STRUCTURE....................7
3.1 ReSP Core 8
3.2 ReSP IPs 9
3.3 ReSP Tools 9
3.4 The compilation system....................10
3.4.1 Compiling a normal file....................10
3.4.2 Compiling a SystemC component....................11
3.5 ArchC: the processor description language....................14
3.5.1 Overview 14
3.5.2 Performed Modifications....................14
3.5.3 TRAP (Transactional-level Automatic Processor generator)....................16
3.6 Integration of the GDB debugger....................17
3.6.1 Standard GDB Stub....................18
3.6.2 Debugging in time....................19
3.6.3 Other Debugging Tools....................19
3.6.4 Extensions to GDB....................20
3.6.5 Current Issues 20
3.7 Profiler 21
3.7.1 Function Profiling....................21
3.7.2 Callgraph 22
3.7.3 Planned Improvements....................22
3.8 Boot-Loader 22
3.9 OS Emulator 23
3.10 SystemC modelling in Python....................27
3.11 Static Platforms 27
3.12 ReSP User Interface....................27
3.12.1 Console 28
3.12.2 GUI 30
3.13 Component Models....................30
3.13.1 UART 30
3.13.2 Interrupt Controller....................31
3.13.3 Timer 31
3.13.4 Cache 31

3.13.5 Processors 32
3.14 eCos Operating System.....33
3.14.1 Porting to the GNU/GCC 4.2 compiler.....33
3.14.2 Porting to ReSP.....33
3.14.3 Modification to enable SMP.....34
3.14.4 Using eCos 34
3.15 RTEMS 35
3.16 Bug Fixing 35
3.17 Dissemination 36

4 FUTURE WORK.....37

5 REFERENCES.....39

1 INTRODUCTION

This document contains the description of the activities carried out during a stage of nine months at the ESA ESTEC research centre in which the author contributed to the development of a simulation platform (*Reflective Simulation Platform* or *ReSP*). The next paragraphs motivate the need for such a tool.

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-spots, allows better usage of the chip surface and provides more possibilities to exploit parallelism. This brought to the definition of Multi-Processor System-on-Chip (MPSoC). The design of MPSoC raises new challenges due to the large design space and tight design and time-to-market constraints. MPSoC are complex devices, and therefore they require 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.

In particular the main difficulties, due to the introduction of MPSoCs in the development and design flow, are:

- The architecture may include more than one application specific processor;
- Inter-processor communication may require networks more sophisticated than a simple shared bus;
- Processing elements cannot be independently designed and optimized: this causes an exponential growth of the design space.

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, granularities, 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 [F. Ziegler et al., 1996[i]]. 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 difficulties 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'' [McDowell et al., 1989[ii]], 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. Moreover, 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.

Another feature that is more and more common in embedded systems is the presence of reconfigurable hardware; this is justified by the fact that requirements for flexibility and adaptivity are gaining importance is such systems. Such requirements could be solved by moving functionalities to software, but the adoption of reconfigurable hardware exhibit higher computing capabilities, storage capacity, and flexibility.

New tools and methodologies are already emerging for the design and the development of ``static" embedded systems; in particular, the modeling at high abstraction levels and the massive re-use of pre-developed IPs are becoming standards. In the context of reconfigurable systems, such techniques would enable the early identification and quantification of the trade-off between area use, configuration time, and power consumption. These are normally explored only after the initial system prototyping, with severe consequences in terms of re-design time and costs in case wrong assumption were made. This is especially true when hardware configuration time is high, as its correct estimation is of critical importance. As a response to these needs, ReSP was extended with capabilities for high-level design space exploration, focusing in particular on devices with embedded Field Programmable Gate Arrays (eFPGAs) components.

ReSP (Reflective Simulation Platform) is a hardware simulation platform, based on the SystemC and TLM, libraries particularly developed to address the just introduced issues; ReSP is primarily targeted to the simulation of Multi-Processor Systems, though it can in general be used for any hardware system. The Python language is also employed in order to easy the interaction between the simulator and the component IPs and in order to simplify the management of the component models.

The rest of this document is organized as follows: first we will present a short overview of what a *virtual platform* is (ReSP is a virtual platform) and why and how it can be useful for the development of mixed hardware/software systems. Then, ReSP is presented in detail in the following sections, concentrating first on a description of the overall structure of the tool, and then on the different ReSP utilities (debugger, profiler, etc.) and on the SystemC IPs developed for the integration with ReSP. At the end we present the current limits of the tool and what are future plans for addressing them. A short tutorial on how to use main ReSP functionalities is also included in this document.

2 BACKGROUND

2.1 Virtual Platforms

According to [ESLDesVer[iii]], a Virtual Platform is a system level model that characterizes real system behaviour. It operates at the level of processor instructions, function calls, memory accesses and data packet transfers, as opposed to the bit-accurate, nanosecond-accurate logic transitions of a register transfer level (RTL) model.

Virtual system prototyping has three principal use cases:

1 System architecture development, analysis, optimization and validation.

2 Software development, validation and debug using co-developed hardware models.

3 Hardware development, verification and debug using co-developed software.

The expected results of these use cases include:

- A system architecture that meets performance and cost goals with optimized HW/SW partitioning and optimized hardware architecture. Optimized hardware architecture would also act as a "golden" reference for RTL implementation.
- Hardware/software (HW/SW) co-development for the production of application software and system software before a real hardware prototype becomes available.
- Faster hardware implementation, verification and debug, and faster HW/SW co-verification and debug, with respect to performing these activities at RTL.
- Easy component and silicon intellectual property (IP) reuse.
- Fast HW/SW integration and debug.

Nowadays we can observe [Murray, 2007[iv]] that there are two distinct users of Virtual Platforms: system designers and SoC designers. System designers emphasize early software development, while SoC designers tend to use the methodology for HW/SW co-development.

We can say that ReSP is an extended Virtual Platform in the sense that it allows modelling at multiple levels of abstraction, going from Transaction Level to register transfer level (RTL). ReSP tries to address all the typical use cases of a Virtual Platform; it poses as a development platform both for software testing and validation, hardware development and optimization and, also, hardware/software co-design.

2.2 Transaction Level Modelling

ReSP can operate ad different levels of abstraction, depending on the level of abstraction of the descriptions of the component models. The preferred modelling style for ReSP is anyway at Transaction Level: this allows good simulation accuracy together with maintaining a reasonable simulation speed. Figure 2-1 (taken from [Gajski et al., 2003[v]]) shows the different modelling styles and abstraction levels at which it is possible to model a hardware system; x-axis in the graph represents computation and y-axis represents communication. On each axis, we have three degrees of time accuracy: un-timed, approximate-timed, and cycle-timed. Un-timed computation/communication represents the pure functionality of the design without any implementation details. Approximate-timed computation/communication contains system-level implementation details, such as the selected system architecture, the mapping relations between processes of the system specification and the processing elements of the system architecture. The execution time for approximate-timed computation/communication is usually estimated at the system level without cycle-accurate RTL (register transfer level) /ISS (instruction set simulation) level evaluation. Cycle-timed computation/ communication contains implementation details at both system level and the RTL/ISS level, such that cycle-accurate estimation can be obtained.

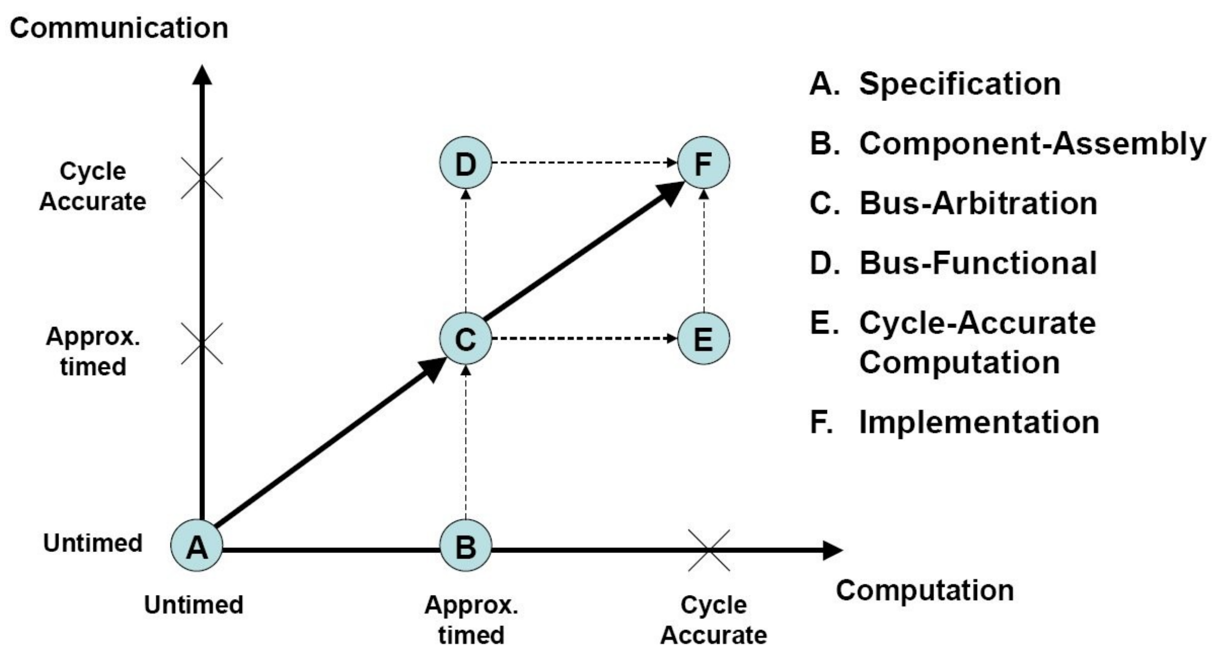


Figure 2-1: Gajski System Modeling Graph

In the transaction-level model(TLM), the details of communication among computation components are separated from the details of the implementation of computation components. Communication is modelled as channels and transaction requests and it takes place by calling interface functions of these channel models. Unnecessary details of communication and computation are hidden and may be worked out later. Transaction-level modelling enables speeding up simulation time, by exploring and validating implementation alternatives at the higher level of abstraction.

Currently, Transaction Level Modelling is enabled in ReSP through the use of the SystemC [OSCI[vi]] library and the OSCI TLM 2.0 Draft 1[OSCIvi] library; a transition to the just published TLM 2.0 Draft 2 library is planned; as a consequence of this transition, a consistent simulation speedup is expected thanks to the possibility of temporarily desynchronizing the component modules with the global SystemC clock.

3 OVERALL STRUCTURE

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* will be 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 will refer to the term *component* to describe any top-level SystemC module included into the framework's database.

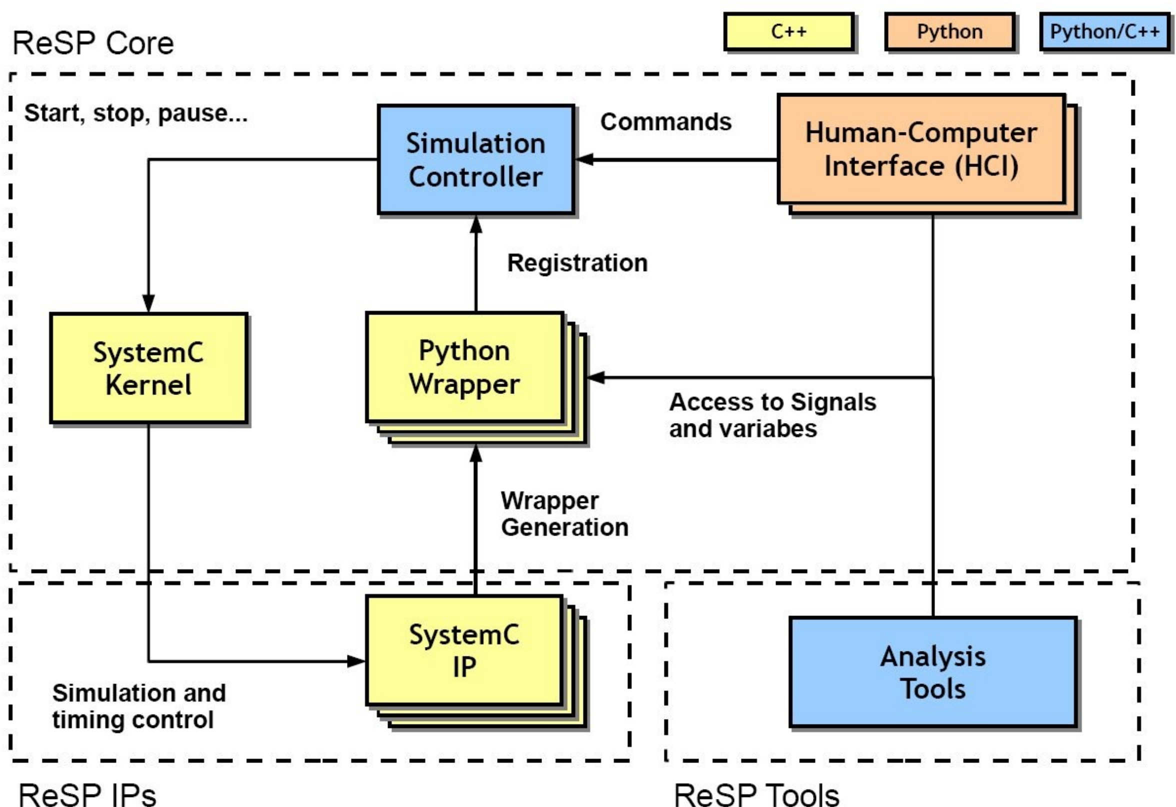


Figure 3-2: Overview of the ReSP architecture

The proposed framework is based on the concept of reflection [Foote at al., 1989[vii]], 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 IPs can be integrated in the system with minimum effort.

Figure 3-2 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.

3.1 ReSP Core

The core of the ReSP architecture is the OSCI *standard SystemC kernel*, as directly released by OSCI [OSCIvi]. 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 it 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 natively present in SystemC; this concept was introduced in [Paulin, 2002[viii]] 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 will be used both for the 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 3-3. 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++ [Py++[ix]] 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 of the 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.

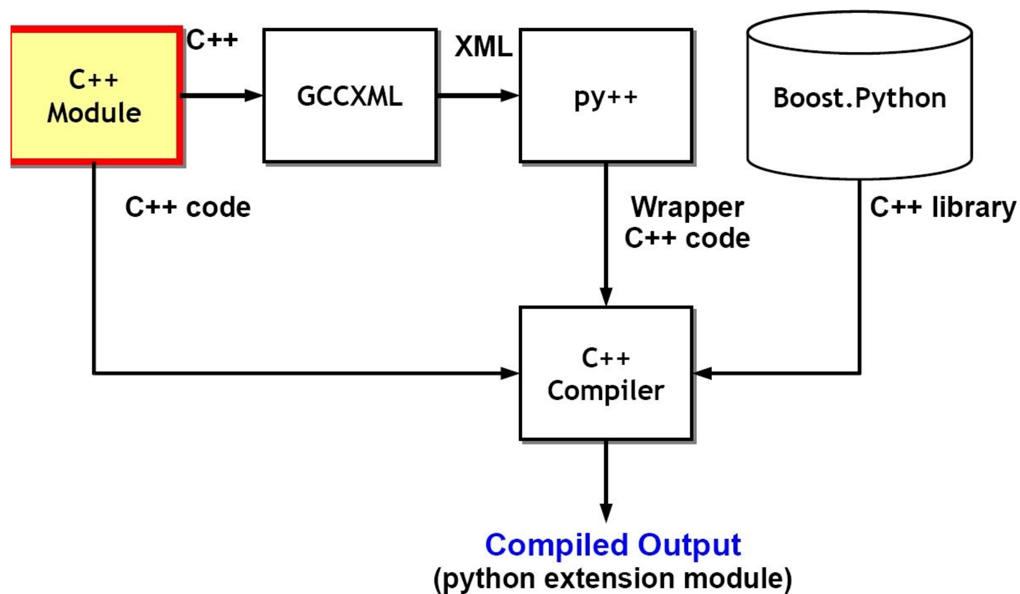


Figure 3-3: Wrapper Generation Steps

3.2 ReSP IPs

One of the peculiarities of ReSP is the capability of easily integrating and interconnecting any valid SystemC component; in fact, as described in the previous section, it is not necessary to modify the components' descriptions since 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 [Rigo et al., 2004[x]] Architectural Description Language; we possess both the functional and cycle accurate versions of the PowerPC, and ARM7 RISC processors. Not fully featured version of the Leon2, NIOS2, MicroBlaze, and MIPS processors are also present.
- *interconnections* in terms of bus and Networks-On-Chip;
- *memory systems* including simple memories and caches (in particular we have the model of the Leon3 L1 cache);
- *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.

3.3 ReSP Tools

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 to 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. Instead, automatic execution mode is used to run in batch mode a sequence of n simulations.

3.4 The compilation system

ReSP compilation system is not based on *make* and *autotools* as most Unix based projects are but, instead, it uses *waf* [WAF[xi]]. This choice brings many advantages:

- **Portable**: Waf only depends on Python which is ported onto most operating systems
- **Fast**: Waf is much faster than autotools
- **Easy to use**: Waf configuration files are written in Python which is mainstream scripting language
- **Easy to install and to distribute**: Waf fits entirely in a single **130KB** redistributable file which does not require any installation whatsoever: this also enable the personalization of waf itself.

Three tools were developed for waf: ***py++***, ***mysql***, and ***mkshared***: the first is used to automatically generate Python wrappers around SystemC components, the second takes care of the creation of the databases necessary for a correct operation of the timed debugger (described in the following sections) and mkshared takes care of the creation of a shared library starting from a static one: this is mainly needed to convert SystemC library to a shared library in order to enable its used from Python. These tools are contained in folder *tools/waf*.

3.4.1 COMPILING A NORMAL FILE

Instead of using Makefiles, waf uses *wscript* files. They are written using the Python language, so any valid piece of Python code can be inserted into them, highly enhancing the capabilities of the compilation system.

```
#!/usr/bin/env python
# Build

def build(bld):
    obj = bld.create_obj('cpp', 'program')
    obj.source='''
        backtracer.cpp
    '''
    obj.includes = '../../src/profiler
../../src/bfdFrontend ../../lib/binutils-2.18/build/include'
    obj.uselib='BOOST OPCODES BFD LIBERTY BOOST_REGEX
BOOST_PROGRAM_OPTIONS'
    obj.name = 'backtracer'
    obj.target = 'backtracer'
    obj.uselib_local = 'BFDFrontend'
```

This wscript takes file *backtracer.cpp* and it compiles it as a C++ (cpp) program called *backtracer*; during compilation the folders listed in *obj.includes* will be examined for include files (these folders will be arguments to the -I option of the compiler). The libraries listed in *obj.uselib* have already been determined at configure time and they will be used for the compilation of this object; also libraries listed in *obj.uselib_local* will be used, but these libraries were not already present in the system, but they have been created during the compilation of the project. For more details on the compilation system, look at the wscript files inside the ReSP sources; in particular look at *trunk/wscript* for details on the configuration process.

3.4.2 COMPILING A SYSTEMC COMPONENT

The wscript which must be written for the creation of the Python wrappers around SystemC (and, in general, C++ components) is a bit more complicated:

```
def build(bld):
    obj = bld.create_obj('pypp', 'shlib')
    obj.source ='''
        simpleMemory.hpp
    '''

    obj.target='SimpleMemory32';
    obj.start_decls = 'SimpleMemory'
    obj.includes='../../../src/mysql ../../../src/systempy'
    obj.custom_declaration_code= """
#include <tlm.h>
using namespace tlm;
"""
    obj.custom_code = """
cls = mb.global_ns.classes(lambda decl :
decl.name.startswith('SimpleMemory') )
```

```
for i in cls:
    i.include()
    i.alias = 'SimpleMemory32'
    try:
        i.variable('memory').exclude()
    except:
        pass
cls = mb.global_ns.member_functions(lambda decl :
decl.name.endswith('transport'))
for i in cls:
    try:
        i.exclude()
    except:
        pass
"""
    obj.uselib='SYSTEMC_H MYSQLPP_H MYSQLCLIENT_H BOOST
BOOST_PYTHON PYEXT'
    obj.uselib_local='systemc'
    obj.include = "scwrapper"
    obj.templates = ['SimpleMemory<unsigned int>']
```

We start by saying that we are creating a shared library *(shlib)*. The option *obj.start_decls* indicates that, for the creation of the wrapper, we examine only the declarations inside the specified namespace. o*bj.custom_code* is the directive which contains the code controlling the creation of the wrappers; this code is composed by a calls to pyplusplus [PY++[xii]] routines. The last two interesting parts are *obj.include* and *obj.templates*: the former specifies all the wrappers which must be included into this one while the latter is used only for templated classes (it specifies the actual instance of the template which should be exported to Python).

The following are basic instructions on how to compile the tool with the default options; more details can be found in the source code or on the ReSP website (www.resp-sim.org):

Go into trunk folder and run ./waf configure using parameters --with-systemc=systemcpath and --with-tlm=tlmpath commands; note that tlmpath must be the base TLM folder (e.g. /home/luke/TLM). Both those folders must be specified using an absolute path. Finally run ./waf in order to compile everything (in case the building process stops with a strange error try to execute again ./waf; in order to run the compilation tasks in parallel you can use the -j option specifying the number of parallel tasks that you want to execute; since this feature is still experimental it may fail: in this case try re-running the compilation without using this option). Some compilation options can be given to the configure command:

- boost-libs : the directory (absolute path) were the binaries of the boost library are installed. You usually do not need to specify it; use it only if the libraries are installed in a non-standard directory.

- boost-includes : the directory (absolute path) were the headers (.hpp files) of the boost library are installed. You usually do not need to specify it; use it only if the headers are installed in a non-standard directory.
- enable-debug : provides the simulator with the capability of debugging simulated software; note that simulation will run slower with this flag enabled.
- enable-profiler : provides the simulator with the capability of profiling simulated software; note that simulation will run slower with this flag enabled.
- enable-tracing : provides the simulator with the capability of printing the trace of the executed assembly instructions; note that simulation will run slower with this flag enabled.
- enable-processor : enables only a subset of the available processors; if none are specified (or if this option is not used) all of them will be compiled. Currently the available processors are arm7 (functional arm7tdmi processor), arm7ca (functional arm7tdmi cycle accurate processor), leon2 (functional leon2 processor), mips1 (functional mips processor), powerpc (functional powerpc 405 processor), ppc405 (cycle accurate powerpc 405 processor); pass to this variable a space separated list of the processors (e.g. --enable-processor=‘‘leon2 arm7‘‘).
- disable-os-emu : disables the emulation of the operating system in the processors: this means that you will need to run a real operating system (such as eCos or RTEMS) on the processors. If the OS emulation is enabled you will be able both to run a real OS or to use the Os emulation, but execution with a real OS will be slightly slower.
- debug-level : specifies the optimization level of the produced code; the valid values are ultradebug, debug, release and optimized.
- arm-compiler : the path of the ARM cross-compiler; the cross-compiler can be downloaded from the trac website hosting the project
- sparc-compiler : the path of the SPARC cross-compiler; the cross-compiler can be downloaded from the trac website hosting the project
- mips-compiler : the path of the MIPS cross-compiler; this cross-compiler hasn't been created yet
- powerpc-compiler : the path of the POWERPC cross-compiler; this cross-compiler hasn't been created yet

For what concerns eCos, three configuration flags are necessary

- ecos-path : the directory of the eCos sources: this configuration option is necessary in order to enable support for eCos source code.
- wxgtk-path : the path of the source code for the wxGTK-2.4.0 library; if this option is not given, the graphical configuration tools will not be built.
- ecos-bin-dest : the destination folder where the tools for the management and the configuration of eCos will be installed; by default they are installed in ˜/ecos-bin.

Few notes on the installation of eCos: the patching and compilation of the eCos related tools will be performed only the first time ReSP is set up; successively, the ReSP build system will check for the existence of the folder specified by the option –ecos-bin-dest: if this folder already exists, no action will be performed. So, in case you want to reinstall eCos, all that is necessary should be

deleting this folder (anyway we suggest also to download new, unpatched, versions of eCos and wxGTK-2.4.0).
For other minor configuration and compilation options simply run ./waf --help at the command prompt. Finally run ./waf in order to compile everything (in case the building process stops with a strange error try to execute again ./waf –nocache). Note that this step takes a long time (10 minutes on a 2.33GHz Dual Xeon PC with 4 GB of RAM). After compilation finishes, in order to test the installation run ./startSim.sh -a architectures/test arch.py in the project main folder (the one which contains the startSim.sh script). Other tests can be fired up by running ./waf -C (this takes a lot of time).

3.5 ArchC: the processor description language

3.5.1 OVERVIEW

ArchC [Rigo et al., 2004x], is an open-source, SystemC based, language that is specialized for processor architecture description. Its main goal is to provide enough information, at the right level of abstraction, in order to allow users to explore and verify new architectures, by automatically generating software tools like simulators and co-verification interfaces. All the processors' models contained in ReSP are described using this ADL.

An architecture description in ArchC is divided in two parts: the Instruction Set Architecture (AC_ISA) description and the Architecture Resources (AC_ARCH) description. Into the AC_ISA description, the designer provides details about instruction formats, size and names combined with all information necessary to decode each instruction. The AC_ARCH description informs ArchC about storage devices, pipeline structure etc. Based on these two descriptions, ArchC (using its *ac-sim* and *actsim* tools) can generate interpreted simulators both in the functional and cycle accurate versions. For more details on the inner working of the ArchC language, refer to [ArchC[xiii]].

Both in order to increase the integration with ReSP and in order to correct some bugs and improve language efficiency, some modifications were performed to the official ArchC version. They are described in the next Section.

3.5.2 PERFORMED MODIFICATIONS

The ArchC language was chosen as the language with which the ReSP processors are described (but keep in mind that any processor model can be used with ReSP, as long as it is written using the SystemC library) because (a) it is OpenSource (b) it is based on the SystemC language and on the TLM library (c) there is a consistent library of processor models described with this language [ArchCModels[xiv]]. Unfortunately the new version of ArchC (version 2.0, which is also the first version based on the OSCI TLM library) has some deficiencies which needed fixing; moreover, in order to achieve a better integration with ReSP, some modifications to the code and additions of functionalities were necessary.

Integration with Profiler, Debugger and Instruction Tracing
Few modifications must be performed to the ArchC *acsim* and *actsim* tools (they are the tools which take the high level processor description and create the simulator code) in order to force the generation of processor code containing the variables and methods necessary for supporting software debugging, software profiling and instruction tracing; these functionalities will be described more in detail in the following sections. After the compilation of ReSP, the tools are inserted in folder *bin.*

Update to the TLM 2.0 Draft 1 library
ArchC 2.0 is still based on the outdated OSCI TLM 1.0 library; the newer, TLM version 2.0 Draft is used in ReSP. The porting involved the change of some files in the *ac_tlm* directory of the ArchC sources; these changes were more syntactic than semantic anyway.

Support for Cycle Accurate processors
ArchC version 2.0 does not yet completely support the generation of cycle accurate processors; most of the necessary functionalities had anyway already been implemented. Changes were made in order to complete the support. Anyway it is still very cumbersome and complicated to correctly create cycle accurate processors, to the point that it is almost impossible to do it. In order to enable this feature a complete rewriting of ArchC would be needed (we are actually making an attempt in this direction with the TRAP tool, briefly introduced at the end of this document). The main problems are related to the handling of the pipeline flush and stall operations; for their insertion consistent modification to the language would be necessary.

Interrupt Handling
The interrupt handling routines originally contained in ArchC were based on synchronous mechanisms: this means that when a hardware component generates an interrupt, the whole system is halted until the processor receiving the interrupt has not completed servicing it. This, of course, does not reflect how a real hardware system behaves. The communication between the processor and the interrupt source has to be split into two phases: first the interrupt gets to the processor (and, in the meanwhile, the system continues its normal operation), then, in a later moment, the processor decides to take the appropriate action (which might even consists in ignoring the interrupt). It is up to the user to write the code which checks if an interrupt has reached the processor or not: this is another limitation of the language, since such an operation could be, instead, automatically performed.

Software Application Loading
ArchC processors already contains the code for parsing an ELF file and loading its content into memory; this operation was also used to initialize the processor registers (setting the stack pointer at the end of the application data, setting the program counter to the program entry point, etc.). This was not suitable for our purposes for two reasons: (1) in a shared-memory multi-processor environment we cannot let each single processor load the application software (there is the risk that each processor overwrites what loaded by the others), but anyway we need a way to initialize the registers of each processor (2) TLM channels can be used only after the SystemC kernel has been started, while we need to load the application before that time: this means that we need to directly

load the executable code into memory, without passing through the TLM channels of the processor.
To solve these problems an external binary loader was written; the size of the program, it start address and entry point are then passed to each processor for the initialization of its internal status, while the machine code of the program that we want to simulate is directly loaded into memory. The code for the loader is contained in folder *src/execLoader*.

The complete source code of ArchC, including the performed modifications, is included inside ReSP (in folder *lib/archc*) and it is automatically compiled when ReSP itself is compiled.

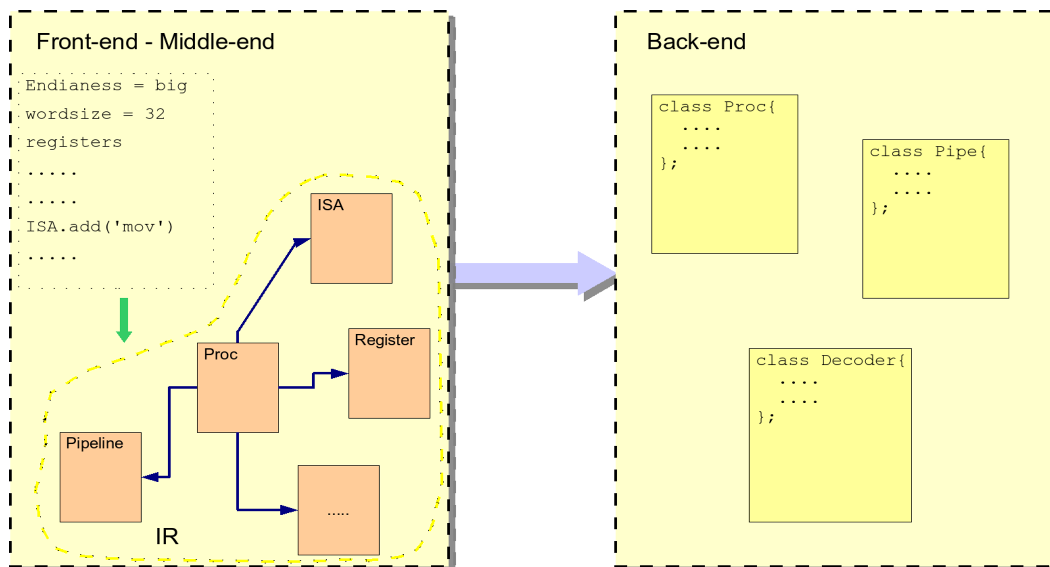
3.5.3 TRAP (TRANSACTIONAL-LEVEL AUTOMATIC PROCESSOR GENERATOR)

Even after the modifications described above, ArchC remains an unsatisfactory tool for the following reasons:

- simulation speed: even the functional processors on a recent computer (Core 2 Duo architectures) are not able to go beyond 2.5 MIPS
- cycle accurate modelling: the support for the generation of cycle accurate processors is completely missing. The language was stretched to enable modelling of the pipeline and of the stall and flush operations for the modelling of the ARM processor. To make the mechanism more generic other patches to the language would be needed. Also the speed of the simulator is very low (more than one order of magnitude slower than the functional processor).
- coprocessor support: ArchC currently has no support for coprocessors; this is necessary since most of the currently used embedded systems (e.g. ARM and LEON) use the coprocessor to execute floating point instructions.
- TLM 2.0 Draft 2 support: ArchC originally supports only the OSCI TLM 1.0 library; this was extended for TLM 2.0 Draft 1 (mainly syntactical changes). Support for Draft 2 would bring consistent benefits in terms of interoperability with other external IPs and in terms of simulation speed. But the changes in the TLM library from Draft 1 to Draft 2 are consistent and they include both syntactic and semantic novelties; so a consistent effort is necessary for the support of the new TLM version.
- Support for architecture peculiarities: there is not particular support for specific features of the different processors such as branch prediction, watchdogs ...
- Unable to model not existing processors: more that the inability to model not existing processors, ArchC lacks the possibility of producing a compiler for such an architecture (thus preventing the use of the created model).

For these reasons we started the design and implementation of TRAP (*Tr*ansactional *A*utomatic *P*rocessor generator), a new tool for the automatic generation of processor simulators starting from high level descriptions.

The tool will be written in Python and it will produce SystemC based simulators. With respect to standard ADL, having the input directly from Python eliminates the need for having an ad-hoc front-end (as shown in the following Figure), thus consistently reducing the development effort.



The main goal of TRAP is the generation of fast instruction set simulators (ISS) both in the cycle accurate and functional flavor; the TLM 2.0 library will be used to model communication among the IPs: as opposed to using version 1.0 this should consistently speed-up simulation and it should guarantee a higher interoperability with the other IPs. When completed, TRAP will enable the generation of RISC, VLIW and superscalar processors; in order to simplify the development process and have a short development cycle, features will be added one at a time, starting with the support for RISC processors, the others following.

At present, the back-end has been completed and carefully tested, and we are concentrating on the specification of the API which will be used to input the high level description of the target processor architecture.

The code implementing the mentioned functionalities is contained in folder *other/cxx_writer* for the back-end and *other/trap* for the front-end – middle-end.

3.6 Integration of the GDB debugger

In order to be effectively useful to software developers, ReSP must integrate some form of debugging mechanism; this way we can have the ability to debug software before silicon availability, using familiar software development and debug environments. For this last reason, and

in order to reduce the development effort, we decided to base ReSP debugger on GNU/GDB [GDB[xv]]. The integration among GDB and ReSP was performed using the GDB *Remote Serial Protocol* [GDB Remote[xvi]] through socket connections.

Debugging concurrent programs is not an easy task; the problem is made even worse if a multi-processor architecture is considered. In addition we desire to enable hardware/software *co-debugging*: this can be useful to hardware designers to test that their hardware models are correct, but also to software architects in order to have a clearer understanding of what is happening inside the system while the software program is running. Enabling all these features implies using more powerful debugging capabilities that the ones provided by a plain use of GDB.

The following Sections present in detail the work done first to correctly communicate with GDB and then to augment its debugging capabilities and functionalities.

3.6.1 STANDARD GDB STUB

GDB, using the *Remote Serial Protocol*, is able to talk with the remote system (an external board connected through the serial port, a program running on another PC connected through sockets etc.) actually running the application under debug; we will call *GDB Stub* the part of the remote system responsible for communication with GDB. GDB is responsible for parsing the debug information associated to the software program and for keeping track of the correspondence between the assembly and C (or C++) instructions. GDB also manages the interaction with the user. On the other hand, the GDB Stub interprets the commands received from GDB, acts by interrogating the system (i.e. the memory, the processor etc.) and by delivering answers back to GDB.
In order to enable GDB support in ReSP we simply had to implement the GDB Stub; this stub was created in order to support multi-processor architectures and provide coordination among the different processors. For instance, when a processor encounters a breakpoint, also the execution of the other processors is halted and the corresponding GDBs are awaken so that they can deal with user commands. The current version of the GDB Stub is implemented in such a way that there is a different instance of the stub for each processor running in the system. This does indeed create problems, as explained later, thus future versions of the stub will be created such that only once instance of the stub exists for the whole system and the user can decide, at runtime, what is the processor that the stub is currently monitoring (this last mechanism is similar to what implemented in the TSIM simulator). The code for the GDB stub is contained in folder *src/softwareDebug/*.

Note that each processor model have different registers and a different way of accessing them; a small module (which we call *processor interface*) has to be written for each processor model and registered in the GDB Stub: this way it is possible to apply the same GDB Stub to multiple processor models. Since the processors are nothing but C++ classes, the registers, memory interfaces etc. are variables, object instances; the role of the processor interface consists in mapping the PC, the registers etc to those variables: when GDB requests a read on the PC (for example) this wrapper will read the appropriate variable inside the processor model. The code for

the wrappers (currently we have developed them for the LEON2 and the ARM7 processors) is contained in folder *src/softwareDebug/procStubs/*.

3.6.2 DEBUGGING IN TIME

Since we are in a simulated environment, the visibility over the system state is wider than the one we would have if we were working on a real physical system. In particular it is possible to have a complete history of all that happened in the system from its start. Using this data we decided to extend our GDB Stub in order to introduce debugging in two dimensions: in the program space (all over program instructions, as done by normal debuggers) and in time.

In order to record all the events happened during the simulation the MySQL [MySQL[xvii]] OpenSource database is used. Commands were added to GDB (using the custom *monitor* command, as shown later) so that it is possible to query the simulated time and so that it is possible to choose the debugging direction (forward or backward in time) and also to jump to a specific simulation instant. For example, a normal watch on variable *a* stops the simulation when the address, where *a* resides, is accessed. If we know that in variable *a* there is a wrong value, it is possible to debug *back in time* and put a watch on *a*: when the simulation is restarted, it will run back in time until last time *a* was accessed.

Another advantage of this approach is that while the developer is debugging the application (and so, in theory, with a normal system, simulation should be stopped) simulation can continue in background (thus real simulation time gets decoupled from simulation time as seen by the developer) and the produced events are logged on the database. When the developer wants to resume execution (for example by issuing a continue command in the GDB console) the GDB Stub will first examine the events logged in the database and then, it will get realigned with the actual simulation. This feature allows to overcome the slow simulation speed that, usually, characterize those systems. Saving the status of all the hardware IPs it is not enough to be able to restart the simulation from an arbitrary point in time: we also need the ability to save and restore SystemC status. This is achieved by a small patch to the systemc kernel. In order to minimize simulation slow-down, the checkpoint of the SystemC status is taken only at a user-defined interval.

Note that not all the described features are completely working yet; in particular there restarting simulation from a generic checkpoint in the past is not properly behaving.

The code for the access to the MySQL database is contained in folder *src/mysql/*.

3.6.3 OTHER DEBUGGING TOOLS

Memory Debugger
This tool can be used, at the end of the simulation, to reconstruct the values in all the memory cells at all simulation instants; it is also able to tell what were all the simulation instants at which each memory cell was modified. These analyses can be produced using the data dumped in the database during simulation.
Backtrace Analyser

During execution the processor is able to produce a trace file with the status of the processor at the end of every assembly instruction. Sometimes, when the developer determines that an error has occurred at a given simulation time, it is useful to determine the sequence of function call that leaded to the instruction being executed at that simulation time (i.e. the backtrace). The backtrace analyser takes in input the trace file produced by the processor and the executable file which was being executed; a time instant *t* also has to be specified. The output of the backtracer consists in the backtrace at the specified time instant *t*.

Tracefile Analyser

This tool is a simple viewer for trace files: given all the trace files produced by the processors of a multi-processor system (each processor produces a trace file), it displays the trace files so that if two events from different trace files are next to each other is because they happened at the same simulation time. A trace file contains, for each instruction executed by the processor, the core dump (i.e. the value of the registers) before and after the instruction execution.

3.6.4 EXTENSIONS TO GDB

In order to allow debugging in time, the set of available GDB commands was extended; in order to call the extensions the GDB command which has to be executed has the syntax *monitor <custom command>* where *<custom command>* can be:

- help: print the list of the available extension commands.
- time: prints the current simulation time.
- status: returns the status of the simulation (the current simulation time, whether it is running backward or forward in time ...).
- forward: set the direction of the simulation forward in time (default).
- backward: set the direction of the simulation backward in time.
- jump n: moves the current simulated time forward (or backward, depending on the simulation direction) of n nanoseconds. This means that the user interaction (breakpoints, possibility of pausing ...) is disabled until we have simulated n nanoseconds.
- jump_abs n: moves the current simulated time to instant n specified in nanoseconds. This means that the user interaction (breakpoints, possibility of pausing ...) is disabled until we reach the desired instant.
- go n: simulates n nanoseconds in the current (forward or backward) direction.
- go_abs n: simulates until the instant n is reached. n is specified in nanoseconds
- awake: awakes the GDBs connected to the other processors so that they can be used for examining the status of those processors. This command is used, for example, when one processor encounters a breakpoint: in order to awake all the other GDBs in the GDB shell of the processors which encountered the breakpoint the awake command has to be issued.

3.6.5 CURRENT ISSUES

During the implementation of the debugger we discovered problems and issues that require consistent modifications to the current implementation; moreover the implementation of the debugging backward in time hasn't been completed yet. In particular:

- *Debugging backwards in time*: the implementation of this feature is not complete yet.

- *Multiple stubs*: the current implementation uses multiple instances of the GDB stub, one for each processor that we want to debug; this means that for each processor we must run d different GDB process. This resulted in a non easy way of controlling the application (e.g. for a 4 processor system there would be 4 GDB processes that the user has to manage). A better way (also used by the TSIM simulator) consists in using only one instance of the GDB Stub connected to all the processors; by using special GDB extensions (through the *monitor* command) it is possible to instruct the GDB Stub about the processor that we desire to monitor. In addition to being more user friendly, this approach also simplifies the coordination among the different processor stubs (well, now there is only one for all the processors).

3.7 Profiler

A profiler is a performance analysis tool that measures the behaviour of a program as it runs; this is necessary since writing applications that make optimum use of computational resources is a challenging problem. Very often, resources are under-utilized or used inefficiently. Such a tool is especially useful on the resource constrained systems used in the embedded domain. In synthesis a profiler produces statistics about the performance of the program in execution. The greatest advantage of our profiler lies in the fact that no instrumentation is performed in the software (unlike, for example, what happens in [gprof[xviii]]). This is an advantage on two sides. First, we are sure that the measured performance really reflects actual performance. Second the program behaviour is not affected; especially in multi-threaded programs, adding probe instructions to a thread, changes its timing with respect to the other threads, and this might, in turn, change the behaviour of the whole program.

The following data is produced: (a) call-graph (b) exclusive time spent in each function (c) global time spent in each function. Such data is computed separately for each processor and for each thread.
The only data necessary for the correct working of the profiler is the current Program Counter (PC) and the current Stack Pointer (SP); the processor passes them to the profiler each time a new instruction enters the fetch stage. So far this information is pushed from the processor core to the profiler: this mechanism is fast, but it requires modifications to the processor core to transfer such information. An alternative consists in the use of the *callbacks* (the implementation is currently not complete yet) which monitor the PC and SP and pull changes from the processor to the ISS; this solution is non-intrusive, but, since the callbacks are written in Python, it consistently slows down execution.
The code for the profiler in contained in folder *src/profiler/*.

3.7.1 FUNCTION PROFILING

In the basic configuration, the profiler is able to produce the following information:

1) *Time spent in each function:* this does not include the time spent in the called functions (i.e. let’s suppose that A calls B: then the time computed will take into

account only the time spent in executing the instructions strictly belonging to A, not the ones belonging to B)

2) *Number of calls to each function*

3) *Time spent in each assembly instruction*: such information is computed for each type of assembly instruction; with this data it is possible, for example, to identify the bottlenecks in the processor ISA (with respect to a particular software program, of course) and consequently modify the hardware architecture. For instance, if our program calls the `MOV` instruction 3 times, spending, respectively, 3ns, 4ns and 2ns during its execution, the total time spent executing the `MOV` instruction will be 9ns.

4) *Number of calls to each assembly instruction*

A more advanced configuration of the profiler (but which requires a considerable higher computational load, which, in turns, negatively affects simulation speed) allows to also keep track of the *total time spent in each function*. This includes both the time spent in executing the instructions exclusively belonging to the function and the time spent executing the instructions belonging to the called functions. This same advanced configuration of the profiler also enables the construction of the call graph.

3.7.2 CALLGRAPH

A callgraph keeps track of how the execution path evolved through the function calls; in this graph each node is associated to a function and each directed edge to a function call. In case function A calls function B, there will be an edge going from A to B in the graph. In addition to building this graph, we are also able to keep track the time at which each function call is made; we, then, annotate the corresponding edge with this information.

3.7.3 PLANNED IMPROVEMENTS

The current implementation of the profiler contains some bugs and limitations, the main one being the execution speed: when the profiler is used the simulation speed drastically reduces. This is mainly due to the fact that many calculations are performed to keep track of the stack of functions that are currently being executed. We can avoid performing this calculations thanks to the routine emulation mechanisms (described in detail in the following Sections); this simplifies the structure of the profiler but, on the other hand, it requires a consistent rewriting.

3.8 Boot-Loader

After the executable file of a program is created by the compiler, we need to load its content in memory in order to be able to execute it. Unfortunately, it is not possible to simply load the whole created file in memory, but it is necessary to correctly extract and interpret the information contained in it. A boot-loader is in charge of this very task. In our boot-loader we employed the GNU/BFD library in order to parse the content of the executable program. This library was chosen

because (a) it is OpenSource, (b) it is already present in most of the linux installations (it is part of the GNU/binutils tools) and (c) it allows the parsing of a wide range of object file formats (though we tested it only with the ELF format).

Our boot-loader parses an executable program in order to extract the following data:

- *Executable Code* and *Global Variables*: these two elements are the most important information saved inside an object file: the first one consists in the machine code which encodes the program behaviour. The second one is composed of the global data: this means the variables which are not allocated in the stack or in the heap (for example, in a C program, the variables declared in a C file outside from any function will be part of the *Global Variables* section).
- *Program Entry point*, i.e. the address of the first instruction which has to be executed
- *Program Start*, i.e. the lowest address occupied by the instructions of the program
- *Loadable data size*, i.e. the size of all the data which have to be loaded in memory for a correct execution of the program; usually this data is composed only by the executable code and the global variables. There might be situation in which this size is bigger than executable code + global variables: in this case, the remaining (size – exec - gblVars) memory has to be initialized by setting all its bytes at 0.

This produced data is used both to initialize the memory and the processors. As explained in Section 3.5.2, this boot-loader replaces the one embedded in the processors generated with the ArchC language.
The code implementing the bootloader is contained in folder *src/execLoader/*.

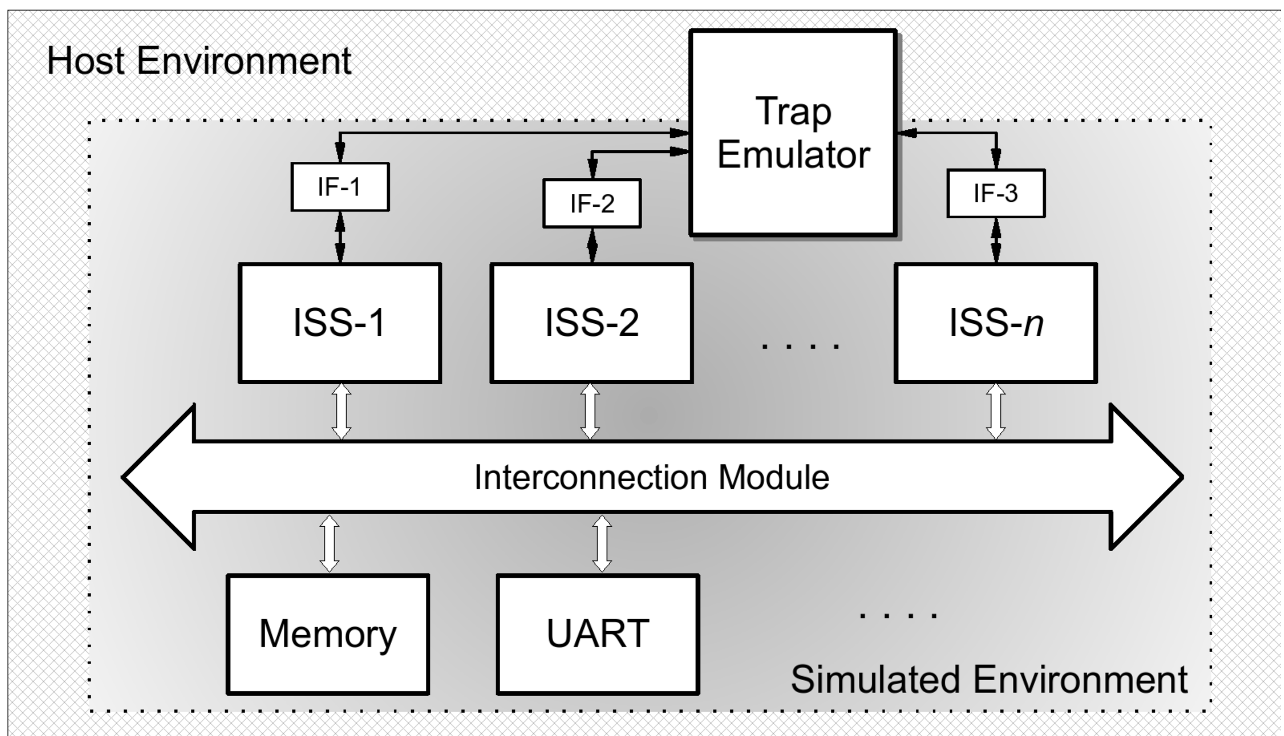
3.9 OS Emulator

System Call Emulation is a technique which allows the execution of application programs on an Instruction Set Simulator (ISS) without the need to simulate a complete OS. The low level calls made by the application to the OS routines (*system calls*, SC) 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 `open` routine to open a file "filename". Such a call is identified by the ISS using the mechanisms described below and routed to the host OS, which actually opens "filename" on the PC's filesystem. The file handle is then passed back to the simulated environment.

Having a simulation framework with System Call Emulation capabilities allows the application developers to start working as early as possible, even before a definite choice about the target OS is performed. These capabilities are also used for ISS validation, by enabling fast benchmark execution. Thus, it is common to find emulation features in instruction set simulators. *Sim-It ARM* and *SWARM* simulate the Service Call (`svc`) instruction of the ARM processor so that the corresponding request is forwarded to the host OS. This requires the emulation mechanism to have knowledge of the various *svc* codes used by the different compilers. ArchC, on the other hand, modifies to the compiler in order to insert jumps to predefined addresses instead of the SC code;

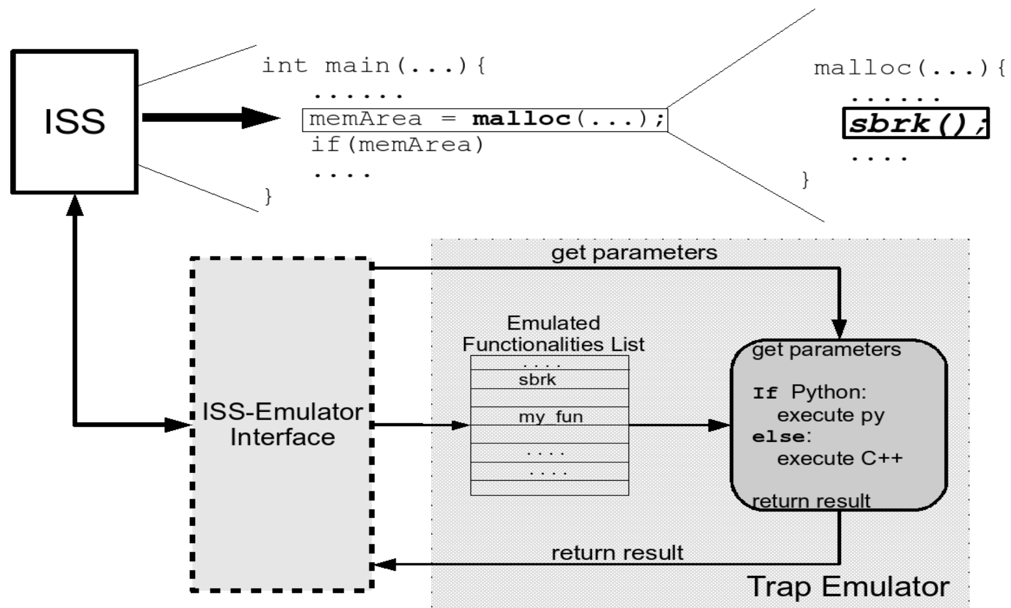
the ISS then takes the appropriate action when one of these particular addresses gets associated to the program counter.

The following Figure shows an overview of our System Call emulation mechanism. Each ISS communicates with one centralized Trap Emulator (TE): this is the component responsible for forwarding the System Calls from the simulated to the host environment. In order to ensure independence between the ISS and the TE, interfaces (IF) are created, and communication between the TE and the ISS exclusively takes place through IFs.

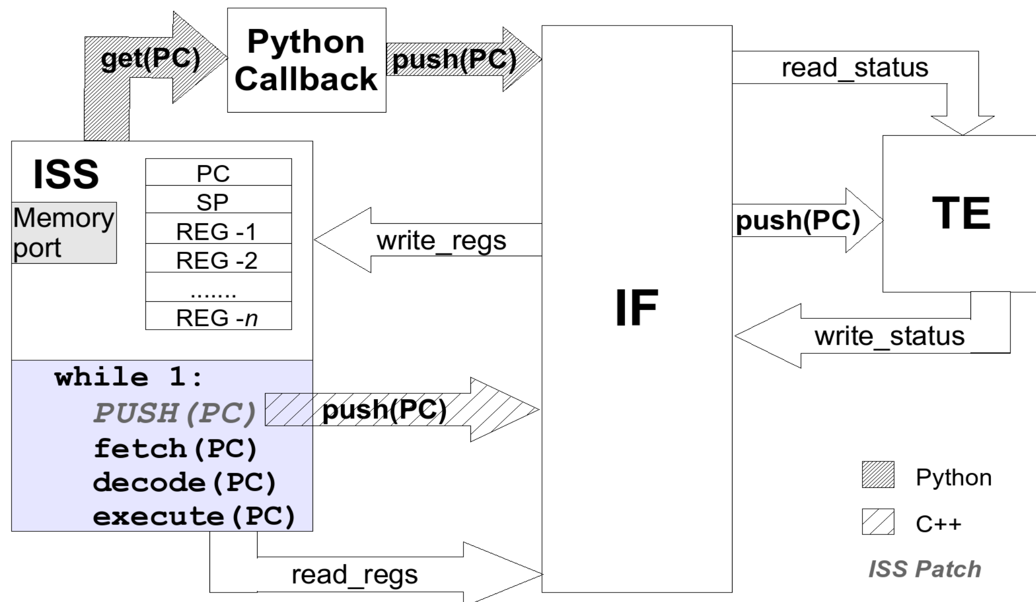


In our approach, instead of identifying the SCs through ad-hoc assembly instructions or special addresses, we use the name (i.e. the *symbol*) of the corresponding routine. When the application program is loaded for execution, the names of the low level SCs (e.g. `sbrk`, `_open`, etc.) are associated with their addresses in the application's binary file. At runtime the ISS checks for those addresses, and when one is found, the corresponding functionality of the TE is called and the SC is emulated on the host environment. The following figure contains a more detailed explanation of the mechanism for the `sbrk` SC: as the library function `malloc` executes, it calls `sbrk`; the TE

checks if the current program counter is to be trapped, and as it recognizes `sbrk`, it stops the ISS execution and performs the appropriate actions.



Communication between the *Emulator* and the *Instruction Set Simulator* is a critical point in the overall design: on one hand it has to be designed so that ISSs can be easily plugged into the system and, on the other hand, it has to be as fast as possible to guarantee high simulation speed. Two solutions were identified: the first one is purely based on the C++ language and it does not exploit any of the *reflective* features of ReSP; the second one uses the Python language to unintrusively access the ISS internal variables.



Writing pure C++ code sacrifices ease of use and flexibility for speed, meaning that the ISS code has to be slightly modified 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.

The second solution is based on the *reflective* properties of ReSP. Instead of having the ISS push each program counter change to the TE, the TE can read it when necessary. This is performed by using *callbacks*, methods which are called when events happen in the simulator. In this case the triggering event is a change in the program counter of each ISS. With Python reflective capabilities and its integration with the C++ language and the SystemC library, callbacks can monitor the ISS without the need for any modification to its source code. Unfortunately, this does not come for free: the callback mechanisms is partly implemented in C++ and partly in Python, and the switching between them is subject to some overhead and consequent simulation slowdown.

The code for the OS emulator is distributed among different folders: *src/SynchManager* contains the synchronization manager (used to deal with `pthread` related system calls) and the part of processor interface for the use by the concurrency manager; *lib/archc/src/aclib/ac_syscall* contains the support for the basic system calls (file opening, closing etc.). The interface used by the normal system calls is contained together with the sources of every processor.

3.10 SystemC modelling in Python

Thanks to the reflective capabilities of ReSP it is possible to describe SystemC components using the Python language; these components can then be connected with other component models written in C++. Of course, with such an heterogeneous system, simulation speed is pretty low; on the other hand development time of the IP models is drastically reduced when writing in Python, so using such mechanisms can help in the first phases of the design process. So, far, due to the use, in the SystemC library, of C++ macros for the declaration of SC_THREAD and SC_METHOD, it is not possible to declare such processes from Python: this means that it is only possible to describe passive components.

3.11 Static Platforms

ReSP is useful for hardware software co-design and for hardware design; it also consistently helps software debugging and tuning through the integration with the GDB debugger and with the profiler. Complete visibility over the internal structure of the hardware components might also help in understanding the evolving of the hardware/software mixed system. Of course, before being able to effectively use ReSP, a developer has to learn how it works and he/she has to install it and the libraries on which it depends. All this effort might not be worth if someone only wants to use the platform to execute software on a fixed architecture.
For this reason we decided to enable the creation of *static platforms*: they are exclusively composed of C++ code and they allow to simulate a fixed architecture (each static platform is like a development board: the hardware components are hardwired in it). The platforms still include the profiler, the GDB stub and all the debugging tool cited in the preceding sections. Moreover the platforms are standalone and they do not depend on any external library. Main files for some static platforms are distributed with ReSP in folder *architectures/static_platforms/*. In order to create a static platform, the option *–static-platform* has to be used at configuration time.

3.12 ReSP User Interface

The ReSP user interface is the mean through which users can interact with the ReSP core, connect components, manage the simulation, etc. There are essentially two different user interfaces: *command line* interface, which runs on the same machine as ReSP itself and it presents the user with a shell (in the Python style) where commands can be entered. The second user interface consists in the *ReSP server*: it is a TCP-IP server through which commands can be sent to ReSP and the results received. Any user interface (as a GUI) can then communicate with this server to interact with ReSP. The advantages of having a socket server are multiple:

- the GUI can be run on a different PC from the one where ReSP itself is being executed
- the GUI can be written in a different language from the rest of ReSP (e.g. there is a draft of a GUI written in Java)

Communication is implemented by encoding the message in terms of character strings (to maximize portability); more details about the communication protocol can be found in the Internals.pdf document downloadable from:
https://trac.elet.polimi.it/resp/trac.cgi/downloader/download/file/3/Internals.pdf

3.12.1 CONSOLE

It is possible to obtain the list of the commands which can be used to interact with ReSP by typing *show_commands* in the console. Here is a brief summary of the list:

`areConnected`
Checks whether source and target are already connected together on ports srcPort and tgtPort. source and target may represent both the instance of the component or its name. On the other hand srcPortName and tgtPortName represents the name of the source and target ports inside those components.

`connectPortsForce`
Method used to connect two ports whose name is not directly visible from python (such as for example ports which are inserted inside a vector): in this case we can't specify the name of the ports which must be connected, but we directly have to pass the instance of the port itself to python. Note that in these situations it is not possible to use the xml input configuration file, but we must directly use the python script to specify the architecture Note that in this case no checks are performed to be sure that the ports are compatible Actually this method may also be used to generically connect two components without checking that the types of the ports we are connections match.

`connectSyscPorts`
Connects together two systemc (NOT TLM) ports; a signal is used to connect together the ports, sourceParams[0] and targParams[0] are the names of the components we want to connect; target and source ports may either be the name of the port or the instance of it. Also the source and target components may either be strings (the names of the components) or the instances of those components sourceParams and targParams represent the parameters which must be passed to the constructor of the source and target components in case they have to be built (so in case source and target are string).

`connectSyscSignal`
Connects together a signal with a systemc port; sourceParams[0] and targParams[0] are the names of the components we want to connect; target and source ports may either be the name of the port or the instance of it. Also the source and target components may either be strings (the names of the components) or the instances of those components sourceParams and targParams represent the parameters which must be passed to the constructor of the source and target components in case they have to be built (so in case source and target are string).

`getCompInstance`
Given the component's name this method returns its instance.

getComponents
Returns a tuple with the names of the components present in the architecture; actually only their name is returned.
getConnected
Returns a tuple with the components connected to component; component may either be the name of the component or the instance of the component.

getSources
Returna a tuple with the component connected to component with master role component may either be the name of the component or the instance of the component.

getTargets
Returns a tuple with the components connected to component with slave role component may either be the name of the component or the instance of the component.

listComponents
Returns the list of the current available architectural components.

printComponents
prints a string containing the list of names of the available the architectural components together with a short description of them.

getAttrInstance
Given the string representing the full path of an attribute inside an object, it returns the instance of the object and the name of the attribute; the instance of the object pointed by path must be present in the current namespace.

getBase
given an object, it checks if in its inheritance chain there is a class whose name is starting with startingName; in case it is found the name itself is returned, otherwise an exception is raised.

getInstance
given the name of a class it builds an instance of that class and returns it; the name itself is returned if no class matching name is found in the global. namespace.

get_real_time_ms
Gests SystemC real time

get_simulated_time
Gets SystemC tim

pause_simulation
Pauses the simulation, waiting for external control.

run_simulation
Runs the simulation for a specified amount of time.

run_up_to

```
Runs the simulation up to the specified simulation time.

stop_simulation
Stops simulation (through a call to sc_stop).
```

Tab completion is enabled in the console in order to facilitate the usage.

3.12.2 GUI

A Graphical User Interface has been developed for the use in ReSP: such a tool eases the use of ReSP and it is useful for attracting potential users. The current version has been developed by a master student of Politecnico di Milano and it is still fairly limited. The GUI has been implemented as an Eclipse plugin using the Java language and it is located in folder *src/gui/eclipse*.

3.13 Component Models

In order to be able to test the functionalities of the simulation platform, we need to model some hardware components and then compose them to form a simulatable architecture. Since the most stable processor model we currently have in ReSP is the ARM processor, we decided to proceed with the implementation of the components contained in the ARM7 PID Evaluation Board [ARM7PID[xix]]. Since there was not a clear explanation of the cache models in the board datasheets, the cache specification was taken from the Leon3 [Leon3Specs[xx]] documentation. This will also help laying the foundations for the implementation of a complete Leon based multi-processor architecture.

The following paragraphs describe in detail how the components currently present in ReSP were implemented. All these components were implemented in SystemC and they use the TLM 2.0 Draft 1 library for describing communication functionalities. All the components are located in subfolders of folder *component*.

3.13.1 UART

This component is necessary in order to enable the communication of the hardware system with the rest of the world. Usually, through the serial interface, the board either communicates with a screen terminal, with a GDB stub for debugging purposes or with both. A real UART controller would command the signals of a serial port, which in turn, for example, is connected with the serial port of a normal PC: on this PC we then convert the signals into characters and we visualize them (using a virtual terminal for instance).
Of course things are a bit different in a software model of a UART controller: since, in this situation, both the source (the UART controller) and the receiver (our virtual terminal) are software components of our system, we simply need to find out a way of making them communicate. Two strategies were devised: *socket* communication and *file based* communication. In the former case the UART simply behaves as a socket server; by connecting to it a client program (such a telnet terminal) we are able to see the characters sent by the hardware system through the serial port. This same mechanism can also be used for sending characters trough the serial port. In the latter case

the UART directly communicates with the linux shell (either by connecting to an already existing terminal or by opening a new one).

A model of the PC16x5x [PC16x5x[xxi]] UART controller was created; this model was chosen both because it is widely adopted, it is part of the ARM7PID development board (used as a reference board for the implementation of the first ReSP architecture) and its drivers are part of the eCOS operating system.
ReSP also contains a model of the UART present in the AT697E Chip; the model of this UART is not complete yet, since only outgoing communication (from the processor to the UART) is supported.

3.13.2 INTERRUPT CONTROLLER

An interrupt controller component is used as a layer between the peripherals issuing interrupts and the processor(s) receiving them. This component is generally used with processors with a limited number of incoming interrupt lines in order to be able to connect multiple devices to the same interrupt line; it might also be used for a more efficient interrupt management (such as a more advanced interrupt masking, etc.). Coherently to what presented so far, we developed a model of the Interrupt Controller contained in the ARM7PID board [PIDIntrContr[xxii]].
A second version of the Interrupt Controller was developed; this version supports multiple processors and, in particular, it allows the management of inter-processor interrupts (one CPU sending interrupts to another CPUs). This last feature was necessary in order to use the *Symmetric Multi Processor* (SMP) configuration of eCos. Also all the Interrrupts (but the inter-processor ones) are routed to only one CPU, no broadcast is supported (again these features are a basic requirement for running eCos).

3.13.3 TIMER

The timer component of the ARM7PID board was implemented. The component is a 16 bit wide counter with selectable pre-scale: the pre-scaler allows either the system clock to be used directly, or the clock divided by 16 or 256 may be used. Two modes of execution are available: free running and periodic.

3.13.4 CACHE

Since no clear specification of the cache module of the ARM processor was found and since we anyway plan to support the Leon3 architecture, we decided to implement the cache model of the Leon3 processor as described in [Leon3Specsxx]; this component includes both the Cache Controller and the separate Data and Instruction caches. Our model shows few changes with respect to the reference manual:

a) we do not support any multi-processor coherency protocol (the original Leon3 cache implements the *snooping* protocol)

b) we support different writing protocols: the original Leon3 cache supports *write through without allocation on miss*, we also support *write through with allocation* and *write back*.

3.13.5 PROCESSORS

ReSP contains many processor models, but only few of them are fully functional; as already described, all the processor models currently available have been developed using the ArchC Architectural Description Language. These components are located in folder *component/processor*. Note that, according to ArchC, for each processor model it is possible to have two different flavours: *functional*, which means that the model is instruction accurate: each instruction lasts a clock cycles; this modelling style does not take into consideration any timing details (e.g. hazards in the pipeline, branch predictions etc.) The *cycle accurate* modeling style emulates not only the functional, but also the timing behaviour of the corresponding processor; cycle accurate models have, anyway, a lower simulation speed with respect to their functional counterpart.

- ***ARM7TDMI:*** we have both the functional and cycle accurate version of the processor, both of them correctly working from a functional point of view and, for what concerns the cycle accurate model, the timing accuracy. The functionality of the processor was assessed by executing many benchmarks on it (including fully featured programs such as ffmpeg, bzip2, etc.). Regarding timing, the validation was performed against SWARM (Software ARM), a bit accurate simulator of the ARM7TDMI.

- ***LEON2:*** we currently possess only the functional version of the processor; for the development of this processor we started from a model of the SPARCV8 processor freely available on the internet. This model did not include the modelling of register windows, different processor execution modes, and interrupts, thus it was not suitable for the simulation of a complete LEON based system; in particular it was impossible to run a real operating system on it. Modification to this model were performed in this direction, but there are still problems and, on some benchmarks, we still experience unexpected errors.

- ***PowerPC405:*** We possess both the cycle accurate and the functional version of this processor; the functional version is freely available on the internet, while the cycle accurate version has been developed by a student of Politecnico di Milano. The functional version only contains the basic processor functionalities (no interrupts, only user execution mode) while the cycle accurate version replicates all the features of the PowerPC405 processor.

- ***MIPS:*** freely available on the internet, it only comes in the functional version and it only includes the basic processor functions (no interrupts, only user execution mode).

- ***NIOS, MicroBlaze:*** developed by students of Politecnico di Milano, they come only in the functional flavour. Both models haven't been completely tested and they are not fully working yet.

3.14 eCos Operating System

A hardware architecture is only a small part of an embedded systems; a complete solution includes also application programming tools, which in turn rely on high-level parallel programming models. According to the market research company, International Business Strategies (IBS), the semiconductor vendors' system-on-chip (SoC) embedded software development effort has grown in absolute terms by more than 4x from the 250nm process node to the 90nm process node, and now constitutes about 56% of the total SoC design effort.

Because of (a) its large support in the OpenSource community, (b) its support for many different hardware architectures and (c) the SMP implementation of its internal routines, we chose to use eCos as the Operating System (OS) of the ReSP platform. Of course, being a generic virtual platform, it does not really depend on a particular version of an OS; but, as we did for the hardware components, we needed something to start with in order to test ReSP, and, as just explained, we chose eCos for the task.

Since ReSP is mainly targeted to the development of multi-processor systems, some modifications were necessary to eCos to enable the SMP functionalities for the ARM target (which, as previously explained, is the one for which we have the most mature SystemC components).

3.14.1 PORTING TO THE GNU/GCC 4.2 COMPILER

Both the core of the eCos OS and the configuration and management tools which come with it are written using the C++ language; unfortunately, the support of this language has consistently changed between the 3.3 GCC series and the 4.2 ones; in particular, some constructs which were supported in the early GCC versions are not supported any more. Since most linux systems now contain GCC version 4.1 or greater and since this compiler version can produce code more optimized with respect to 3.x versions, we decided to support GCC 4.2 for the compilation of eCos. This required modifications to the code in order to correct bugs which prevented compilation.

3.14.2 PORTING TO RESP

Not many changes to eCos were necessary in order to run it on ReSP: this because ReSP models the ARM7PID architecture, which is already supported by eCos. The main modifications were:

- Change of the *configuration scripts*, in order to create a new eCos target (ReSP): by choosing this target eCos will be configured for the architecture modelled in ReSP (which, as just said, is very similar to the ARM7PID).

- Change of the *linker script* for mapping the application at the beginning of memory and for taking into account the lack of a ROM memory.

3.14.3 MODIFICATION TO ENABLE SMP

This phase was much more complicated than the previous one: eCos does not fully support SMP yet: the only target which officially supports Multi-Processors is the i386. Gaisler also performed a porting of eCos to the Leon3 architecture, including SMP support. Unfortunately he worked on version 1.0 of eCos (while we wanted to use the most recent eCos version, 2.0); moreover we have not completed yet the implementation of the component models for the Leon3 target architecture. As a consequence, in order to test the multi-processing capabilities of ReSP, we decided to modify eCos in order to add SMP support to the ARM target. The modifications performed are:

- *Boot sequence*: the boot sequence of a multi-processor system is completely different from the one of a single processor; the different cores have different boot sequences, usually the initialization of the OS is performed by just one processor (called the main processor) and the others can start only after this phase has terminated. The implementation of such functionalities required almost a complete rewrite of the code implementing the boot sequence.
- *Synchronization*: in a single processor system, exclusive access to a critical section of code can be enforced by simply disabling interrupts during the execution of that section; this way no context switch happens, so we are sure that no other thread can enter the same critical section. Disabling interrupts is not enough in a multi-processor system; in this scenario special processor instructions must be used to implement synchronization mechanisms. Every processor usually has its own set of such instructions, so support for the ARM synchronization instructions was added.
- *Stack Location*: in eCos each thread of execution has its own stack so, in this respect, there is no difference between single and multi processor systems. On the other hand, each processor has to reserve special memory areas for the management of the interrupt stacks; this means that, for each processor, the addresses of these memory areas have to be computed, also making sure that there is no overlapping among them.
- *Interrupt Control*: as already hinted above, in order to have SMP support in eCos there is the need of issuing inter-processor interrupts; support for our Multi-Processor interrupt controller was added.

3.14.4 USING ECOS

There are essentially two different ways of using eCos with ReSP: the first one consists in using an already precompiled version of the OS. It is available in folder *software/os/eCos/configurations*; once you have written you application use the command line:

```
g++ yourProgram.cpp -nostdlib -ltarget -lm
-Lsoftware/os/eCos/configurations/ecos_arm_install/lib/
-Tsoftware/os/eCos/configurations/ecos_arm_install/lib/target.ld -Isoftware/os/
eCos/configurations/ecos_arm_install/include
```

The second option consists in downloading a fresh copy of eCos and, then, patching, configuring and compiling it in order to produce the desired Os configuration and BSP; fortunately in order to do this, ReSP provides configuration and compilation options. See Section 3.4 for an accurate description.
Once you have successfully compiled the application, you can use the file *architectures/eCosSMP.py* to instantiate and architecture compatible with the created eCos image.

3.15 RTEMS

Real-Time Executive for Multiprocessor Systems, is a multi-threded kernel supporting Multi-Processor environments using the Message passing programming paradigm. RTEMS supports execution on LEON based environments and, as such, it was used for the validation of a LEON2 based platform.
In order to compile your application for RTEMS and to execute it on ReSP the following steps should be followed:

- Download, either from Gaisler Research's website or from the website of ReSP (www.trac.elet.polimi.it/resp) the following files:
  - *sparc-rtems-4.8-gcc-4.2.2-1.1.0a-linux.tar.bz2*
  - *rtems-4.8-1.1.0a-src.tar.bz2*
  - *mkprom2-2.0.6.tar.gz*
- Decompress those files inside the */opt* folder
- Add to the search path of your system (the PATH environment variable) folders */opt/rtems-4.8/bin/* and */opt/mkprom2/*.
- Compile your application:
  sparc-rtems-gcc -msoft-float -qleon2 -g -O2 myApp.c -o myApp
- Creation of the ROM image:
  mkprom2 -leon2 -nocomp -V -msoft-float -qleon2 -g -O2 myApp -o myApp_prom
  In this step remember to use insert in the mkprom2 command all the flags passed to gcc for the compilation of the application
- Use the configuration file *architectures/rtems.py* to load the appropriate architecture.

3.16 Bug Fixing

A considerable effort was spent in looking for bugs and, consequently, correcting them. This, sometimes, required a complete rewriting of portions of code.

In particular, we concentrated on the tuning and optimization of the build system; this element is central for a correct operation of the simulation platform since it is responsible of the creation of the Python wrappers around the C++ components (thus it is responsible for guaranteeing the interoperability among C++ and Python). A consistent contribute was given to the development of the Py++ and Waf tools, which we use, respectively, for the creation of the wrappers and for the compilation of ReSP.

3.17 Dissemination

In order to favour the utilization of ReSP both internally inside ESA and in the OpenSource community, the following activities were performed:

- Submission of a paper with the title *Event-based Monitoring and Debugging of Heterogeneous MPSoCs in a Co-Design Environment* to the ICCD 2008 conference.
- Preparation of the camera ready version of the paper *ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration* which will be presented and published to the ASP-DAC 2008 conference.
- Sumbission of a paper with the title *ReSP: a design and validation tool for data systems* to the DASIA 2008 conference.
- Sumbission of a paper with the title *Concurrency Emulation and Analysis of Parallel Applications for Multi-Processor System-on-Chip Co-Design.* to the CODES+ISSS 2008 conference.
- Sumbission of a paper with the title *High-Level Modeling and Exploration of Reconfigurable MPSoCs.* to the AHS 2008 conference.

4 FUTURE WORK

At the current state of development, ReSP contains many features and functions, but much work still has to be performed to bring the tool at a stage where it can be successfully and easily used and extended. In particular, we foresee the following possible activities:

- ***Improvement of the profiler***: as explained in detail in the previous sections, currently the software profiler is very slow, due, mainly, to the computations necessary for determining the function call stack active at each instruction. It might be possible to improve the speed and the accuracy of the computed results by only activating the profiler when there is a function call or a return from a function, for example.

- ***Improvement of the debugger***: again, as previously underlined, we are using a separate GDB instance for each processor under debug: this complicates synchronization among the different GDB instances and it also make the utilization of the tool difficult (in case we have 16 processor, we need to have 16 GDB windows opened). We propose to enhance the tool by using a single GDB instance which is able to control all the processors at the same time, switching from one processor to another using GDB extension commands. Also, the implementation of the mechanisms for timed debugging is not complete yet, so the GDB stub should be modified to complete the support for this feature.

- ***Refactoring of the ReSP-Processor interface***: as hinted in the previous Sections, we are currently using a particular processor interface to connect GDB to each processor, a particular interface to connect the profiler to each processor, another to connect the Operating System emulator to each processor and finally there is an interface to connect the concurrency emulator to each processor. The need for all these interfaces consistently complicates the addition of a new processor model to ReSP. We propose to unify all the interfaces in a single one.

- ***Adoption of the TLM 2.0 Draft 2 (or more recent) standard library:*** Currently ReSP can work with components written either using pure SystemC or SystemC and TLM 2.0 Draft 1 library. The TLM 2.0 Draft 2 contains many improvements with respect to the previous version, both in terms of functionalities and in terms of speed. The speed increment is obtained both by a more careful design of the library and by the introduction of new programming paradigms (i.e. the possibility of partially decoupling each component from the SystemC kernel). Adoption such a library implies few modifications inside ReSP itself, but consistent modifications to the IPs, in particular for the active IPs (i.e. the ones that, like processors, initiate communication).

- ***Improvement of the GUI***: the Graphical User Interface currently contains some bugs and it enables only a limited subset of ReSP features.

- ***Addition of features for automatic design space exploration***: embedded systems are characterized by many parameters which determine their efficiency; this aspect is particularly visible in Multi-Processor systems. In order to best configure the system for the execution of the desired task, it is necessary, in principle, to test and evaluate all the possible configurations. We are currently introducing automatic systems which test the different parameter values, also trying to minimize the number of examined configurations, in order to find the optimal system with respect to the application which needs to be executed on it.

- ***IP development***: IPs are the necessary building blocks for being able to successfully model and simulate a hardware platform. As shown in the previous Sections, ReSP currently possesses some SystemC IPs, but most of them are lacking features or they are not properly working. In particular we are missing the IPs which would enable the modeling of the hardware systems used at the European Space Agency. Among them we should concentrate on the models of the LEON2/3 processor and of the AMBA (AHB and APB) interconnection.

- ***Debugging and optimization:*** most of ReSP tools and components still lack adequate verification and debugging. Moreover, for the present activity, we did not concentrate on the optimization (in terms of speed, memory usage, etc.) of ReSP functionalities.

5 REFERENCES

[i] [F. Ziegler et al., 1996]: F. Ziegler et al., **Terrestrial cosmic rays and soft errors**, *IBM Journal of Research and Development*, 1996
[ii] [McDowell et al., 1989]: Charles E. McDowell and David P. Helmbold, **Debugging Concurrent Programs**, *ACM Comput. Surv.*
[iii] [ESLDesVer]: http://electronicsystemlevel.com/
[iv] [Murray, 2007]: Bill Murray, **Virtual platforms - a reality check**, *http://www.scdsource.com/article.php?id=59*
[v] [Gajski et al., 2003]: Lukai Cai and Daniel Gajski, **Transaction Level Modeling: An Overview***, CODES+ISSS'03*
[vi] [OSCI]: Open SystemC Iniziative, **www.systemc.org**
[vii] [Foote at al., 1989]: B. Foote and R. E. Johnson, **Reflective Facilities in Smalltalk-80**, *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*
[viii] [Paulin, 2002]: P.G. Paulin, C. Pilkington and E. Bensoudane, **StepNP: A System--Level Exploration Platform for Network Processors**, *IEEE Design and Test of Computers*
[ix] [Py++]: C++/Python Interfacing: pyplusplus, *http://www.language-binding.net*
[x] [Rigo et al., 2004]: S. Rigo, G. Araujo, M. Bartholomeu and R. Azevedo: **ArchC: A SystemC-Based Architecture Description Language**
[xi] [WAF]: http://code.google.com/p/waf/
[xii] [PY++]: http://www.language-binding.net/pyplusplus/pyplusplus.html
[xiii] [ArchC]: **The ArchC Architecture Description Language**, *http://downloads.sourceforge.net/archc/ac_lrm-v2.0.pdf*
[xiv] [ArchCModels]: *http://archc.sourceforge.net/index.php %3Fmodule=pagemaster&PAGE_user_op=view_page&PAGE_id=8&MMN_position=17:17.html*
[xv] [GDB]: http://www.gnu.org/software/gdb/
[xvi] [GDB Remote]: http://developer.apple.com/documentation/DeveloperTools/gdb/gdb/gdb_33.html
[xvii] [MySQL]: http://www.mysql.org/
[xviii] [gprof]: http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html
[xix] [ARM7PID]: **ARM Target Development System**, Document Number DUI 0061A, http://infocenter.arm.com/help/topic/com.arm.doc.dui0061a/DUI0061A_dev_ug.pdf
[xx] [Leon3Specs]: http://gaisler.com/products/grlib/grip.pdf
[xxi] [PC16x5x]: http://www.national.com/pf/PC/PC16550D.html
[xxii] [PIDIntrContr]: **ARM Reference Peripheral Specification**, Document Number DDI 0062D