High-Level Modeling and Exploration of Reconfigurable MPSoCs

Giovanni Beltrame
European Space Agency
Keplerplaan, 1
Noordwijk, The Nederlands
Giovanni.Beltrame@esa.int

Luca Fossati Donatella Sciuto
Politecnico di Milano
P.zza L. da Vinci, 32
Milan, Italy
{fossati,sciuto}@elet.polimi.it

Abstract

This paper presents a novel methodology for modeling partially dynamic reconfigurable hardware at transaction-level. We cover the lack of tools and mechanisms for the design of reconfigurable logic at system-level and for the exploration of the different configurations of such architectures. The presented mechanisms have been implemented in ReSP, a transaction-level simulation platform especially targeted to multi-processor embedded architectures. In our methodology we allow the definition of the reconfigurable functionalities through scripted languages, therefore the switching of any software function for a hardware one, the modeling of the configuration delays, area use, etc., can be easily performed (even at runtime). Overall, the methodology enables a powerful exploration of the system's functionalities by switching between hardware and software components. Results show that our solution brings flexibility, accuracy, and fast, compared with RTL solutions, alternatives evaluation in the development and design of dynamically reconfigurable embedded systems.

1 Introduction

Nowadays, Multi-Processor Systems-on-Chip (MPSoCs) are a hot research topic. As all industry is moving towards multi-core solutions, the research community is trying to extend the applicability and the performance of MPSoCs, for example for time- or safety-critical applications. These applications have longer lifespans than consumer products, and they require a certain degree of flexibility and adaptivity when deployed in the field. On the other hand, the cost of developing new devices is growing exponentially, and companies are trying to obtain more general solutions to be applicable to larger ranges of products. These new requirements in flexibility and adaptivity translate in new constraints for the software and hardware designers, who also have to strive to reduce time-to-market and costs of product development. A popular solution to these problems consists in the exploitation of reconfigurable logic: this introduces more flexibility in the system without excessively sacrificing performance (as characteristic of pure software solutions).

In general, reconfigurable architectures exhibit high computing capabilities, storage capacities, and flexibility. These architectures often integrate a number of fixed components, including digital signal processing (DSP) and microprocessor cores, custom hardware, and distributed memory modules. In order to be able to successfully implement, deploy and use such complex embedded systems, there is the need to identify, formalize, and implement new concepts, methods and tools. This is especially true if, in addition to the traditional requirements, runtime partial reconfiguration of such devices is taken into account.

New tools and methodologies are already emerging for the design and 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.

In this paper we present a methodology for the high-level design space exploration of reconfigurable platforms, focusing in particular on devices with embedded Field Programmable Gate Arrays (eFPGAs) components. Our methodology uses ReSP (Reflective Simulation Platform), a transaction-level simulation plat-

978-0-7695-3166-3/08 $25.00 © 2008 IEEE
DOI 10.1109/AHS.2008.15

form based on SystemC and TLM libraries, and on the Python programming language. ReSP, which is mainly targeted at the modeling and simulation of Multi-Processor Systems-on-Chip, has been extended to deal with dynamically reconfigurable architectures. The integration of Python in the SystemC library increases modeling capabilities, as functions mapped to reconfigurable logic can be described using both Python and SystemC and they can be inserted and modified at any time during simulation, with simple scripting commands.

The main contribution of this work consists in the introduction of easy means for examining and evaluating the partitioning of the system functionalitites between software and reconfigurable hardware accelerators. During simulation, the area occupied by the various functionalities and the reconfiguration time are taken into account, providing fast evaluation of alternative configurations and reconfiguration scheduling policies.

Experimental results show that the exploration of a sample proof-of-concept application can be easily performed, and that the system designer can easily test new partitioning strategies, considering different delays and reconfigurable block sizes. In addition, our experiments prove that the use of Python provides extreme flexibility, with acceptable simulation time overheads.

The rest of the paper is organized as follows: Section 2 describes some of the most important works related to modeling reconfigurable systems; Section 3 focuses on the concepts of system-level and transaction-level design and introduces ReSP; Section 4 shows the extensions and improvements applied to the simulation platform to deal with reconfigurable systems; finally, Section 5 details the experiments performed to validate the proposed methodology; and Section 6 draws some concluding remarks.

2 Related Work

Several works (such as [6, 7]) introduce dynamic reconfigurability in the context of embedded systems; most of them present design environments and methodologies which facilitate development, mainly concentrating on hardware-software task scheduling and partitioning. Moreover, the majority of these approaches are applied only during the last phases of the design and only few of them consider system-level issues.

The authors of [12] present a methodology for modeling dynamically reconfigurable architectures at system-level using SystemC. They aim at converting existing static Intellectual Property (IPs) into instances of a special component, which models the *dynamically reconfigurable fabric* (DRCF). They do not allow setting reconfigurable functionalities at runtime, but only during compilation; moreover, the created DRCF is not presented in the context of a complete system architecture. A follow-up to this work was carried out in [13], by developing a performance estimation approach for Systems-on-Chip (SoCs) featuring reconfigurable logic. In this context, the configurable components are used as hardware accelerators. A modified preprocessor of the SUIF compiler is used to estimate which functions are to be executed in software and which will be implemented in the reconfigurable fabric.

The SystemC-based framework *Perfecto* [10] aims to perform exploration of different reconfiguration alternatives and to determine system performance bottlenecks. A simple SystemC architecture (bus, processor, memory and reconfigurable elements) is modelled to evaluate the results of hardware/software partitioning, scheduling, and placement algorithms. Their system does not consider multi-processor systems, and the internal details of the work are not described. ReChannel, an extension to the SystemC library, is presented in [14]. ReChannel allows modifications to the system's module topology during simulation, something not natively possible with SystemC. With this feature, the authors can use unmodified static IPs as reconfigurable elements and connect them together using ReChannel. In this context, runtime reconfiguration simply means changing the data flow among the reconfigurable element models. With respect to our work, the authors do not explore solutions using Transaction-Level modeling (TLM) and the presented mechanisms are not inserted in a System-on-Chip environment. Brito et al. [5] approached the problem by modifying the SystemC kernel; custom special SystemC instructions are provided in order to inhibit the execution of the SC_THREAD or SC_METHOD routines of reconfigurable elements not present in the current system configuration. No measures of the simulation speed with the modified SystemC kernel are taken, and routing of communication to the reconfigurable elements is also not considered.

An approach based on the use of function pointers is devised in [3]. The main focus of the authors is around the modeling of the reconfiguration mechanism, and its integration with a simulation environment featuring a general purpose processor (GPP). A limitation of such an approach, compared to the one proposed in this work, consists of the fact that the partitioning of the functionality among hardware and software components has to be done at compile time.

In this paper we address these issues, considering the first phases of the development of an embedded recon-

figurable architecture. We use Transaction-Level Modeling, the SystemC library and the reflective characteristics of ReSP to easily explore different assignments of the system functionalities between software and reconfigurable hardware. This allows the designer to achieve an optimal initial partitioning in a short time.

3 System Level Modeling

When considering a typical system-level design flow. as shown by Figure 1, proper abstraction and modeling are key to efficient and accurate estimation and successful design space exploration [8]. The initial specification of the system is given at a high level of abstraction and then, through iterative refinement, it is lowered to physical implementation, adding more details to the design model with every step. The Reflective Simulation Platform [4], on which the proposed work is built, creates an environment which helps in the refinement process.

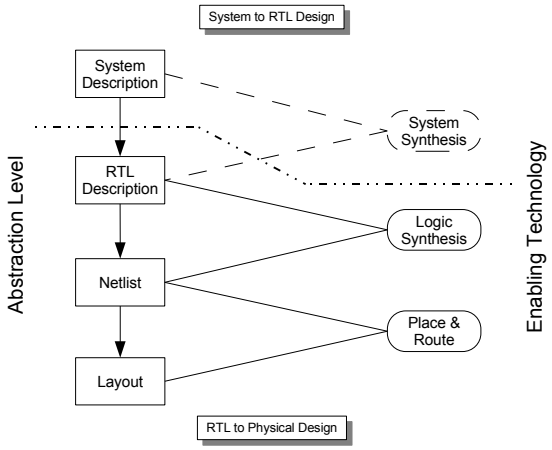

Figure 1. Typical hardware design flow with enabling technologies

The basic 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 analyses, such as design space exploration or reliability assessment. To achieve this goal, ReSP starts from SystemC and the OSCI Transaction Level Modeling library [15] and provides a non-intrusive framework to manipulate SystemC and TLM objects. 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 library of SystemC modules. In the following, we will refer to the term *component* to describe any top-level SystemC module included in the framework's library. The proposed framework is based on the concept of reflection [9], that allows ReSP to observe and control every C++ or SystemC element (variable, method, etc.) specified in any component. This means that, for example, the simulator can inspect a component and determine all of its communication ports without the need to have any compile-time knowledge about them. This means that standard SystemC TLM IPs can be integrated in the system with minimum effort. In this work, we exploit and extend the ReSP's characteristics and tools to include reconfigurable hardware, adding new elements to ReSP's IP library.

4 Design Methodology for Reconfigurability

To describe our approach, we need to define the elements that constitute our model of a reconfigurable architecture. We assume that the reconfiguration (as it happens in many reconfigurable embedded architectures [6]) is software-controlled, and that the reconfigurable blocks are available to the system in the form of embedded FPGA blocks, connected to an on-chip interconnection (a bus or Network-on-Chip). Figure 2 shows the main components of our platform:

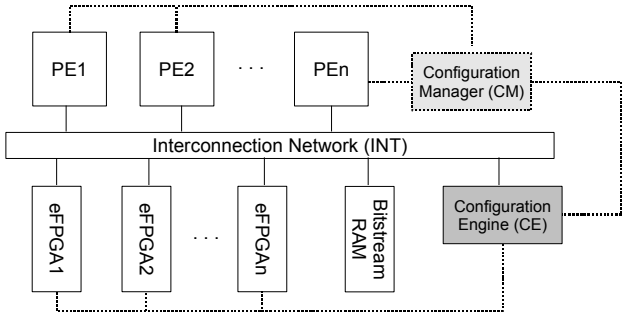

Figure 2. The model of reconfigurable platform used in this work. Dashed lines represent components that exist only for simulation purposes

PEs As this work targets heterogeneous, multiprocessor systems, our platform has a number of Processing Elements, that represent general purpose (or specialized) instruction processors.

eFPGA The embedded FPGA modules represent the reconfigurable elements of the architecture: these

are connected directly to the interconnection network, and they have an internal communication architecture, in case they are large enough to host more than one functionality.

RM The Reconfiguration Manager is a pure modeling component, meaning that it has no counterpart in the real physical architecture. Its purpose is to intercept the requests for the hardware-implemented functionalities and to divert them to the Configuration Engine (CE). Section 4.1 provides more details on this module.

CE The Configuration Engine is a model of the physical component that takes care of performing the actual reconfiguration (e.g. the ICAP module on Xilinx FPGAs).

INT The interconnection network is left unspecified, and can be implemented as a bus, Network-on-Chip, crossbar, etc.

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

4.1 The Reconfiguration Manager

The Reconfiguration Manager (RM) is a C++ component that exists uniquely in the frame of the simulator, i.e. it has no real hardware counterpart. Its purpose is to *model* the adaptive behavior of a system for design space exploration, without enforcing any particular implementation of the reconfiguration mechanism, that is instead modeled by the Configuration Engine. In order for the RM to be able to work, we must place some assumptions on the structure of the PE models. In particular, we suppose that the PEs are modeled as Instruction Set Simulators (ISSs), or, at least, that they are capable of determining which function, procedure, or method they are executing at any time. A simple interface, implemented by the designer, is defined to allow access to the PEs models by the RM. In our implementation, the PEs are SystemC-based ISSs that are interfaced to the RM using C++.

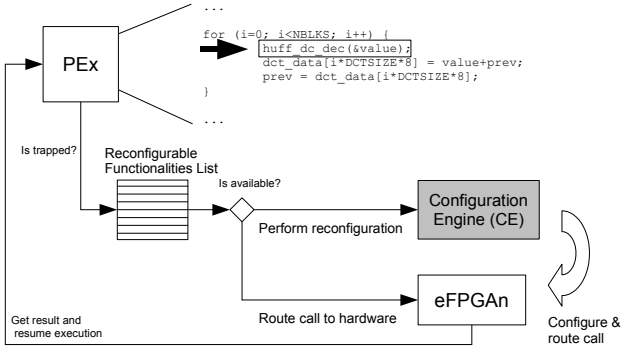

Figure 3. Behavior of the Reconfiguration Manager: it intercepts function calls and routes them to the hardware models

At every instruction issued (or function executed) by the ISS, the RM checks if either the issuing address (or the function identifier) is contained in a list of elements marked for hardware implementation. In this case, the function is trapped and the execution of the PE is blocked. The RM takes control and communicates with the CE: if the requested function is available in hardware, the RM routes the communication through the processor interface to the interconnection hardware and, from here, to the appropriate eFPGA module. When the computation ends, the results are sent back to the PE (e.g. as a memory pointer, or as the function's return value). If the function is not available, the RM issues the appropriate commands to the CE to start the reconfiguration process. In this way, the communication of the hardware reconfiguration modules with the CE and the PE is correctly modeled in detail. There is no need to modify the software running on the ISS, leaving the reconfiguration activity completely transparent to the software developer, except for its delays. This behavior is depicted in Figure 3.

As the RM is initialized, it carries no information about which functions should be marked for hardware implementation. Function registration and deregistration are totally dynamic, and they can even be performed during simulation. To do so, we exploit the reflective characteristics of ReSP: using Python scripts, the designer can define which functions will be mapped to hardware elements. In our implementation, when a given application's object code is loaded, all its symbols and associated addresses are recovered from the executable file. With this data, at any time during simulation, the designer can mark a function for in-

terception by the RM by specifying its name and a reference to a C++ or Python routine implementing the conrresponding reconfigurable module.

Summarizing, the RM is the module which decides whether a given functionality should be executed using dedicated reconfigurable hardware or software. This element communicates directly with the Processing Elements and with the Configuration Engine in order to trigger the computation of a functionality in hardware and (if necessary) the reconfiguration process.

4.2 The Configuration Engine

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

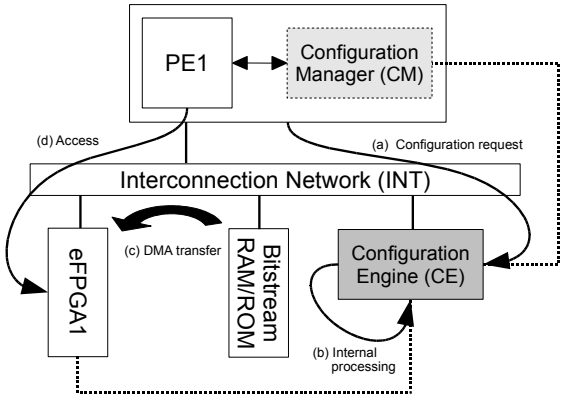

Figure 4. Behavior and delays modeled by the Configuration Engine

The communication between the PE and the CE is modeled using the TLM2 draft1 standard payload interface: communication starts with a request (which specifies the identifier of the function to be called in hardware) generated by the RM and sent through one of the PE's outbound ports. Once the transaction is routed to the CE by the communication network, the CE checks if the specified function is already implemented in one of the eFPGA blocks. If it is, it returns the address space associated with the hardware component, and the RM performs the access to the hardware module. If the function is not implemented, the CE recovers the reference to either the Python or SystemC implementation of the function *outside* the simulation space, by direct communication with the RM. The CE then selects a block (with random or LRU substitution policy), emulates the DMA transfer of the bitstream, and passes the implementation reference to the eFPGA block, that will in turn model the reconfiguration process delay. More formally, this behavior is described in Algorithm 1.

```
   input  : A function f
   output: An address space to where the function
           is mapped
1  if is_configured(f) then
2  |  return(get_address_space(f))
3  else
4  |  wait(internal_delay)
5  |  block = block_available(module.size)
6  |  module = get_module_reference(f)
7  |  transfer(f.bistream,block)
   |  configure_block(module,block)
8  |  return(block.address_space)
9  end
```

Algorithm 1: The CE behavior in pseudocode

The CE delay is implemented as a call to the SystemC `wait()` statement, but it can be easily modified for TLM2 draft2 loosely timed modeling style, by substituting line 4 with an update to the transaction's annotated delay. Concerning sizes and the availability of reconfigurable blocks in the eFPGA model, these are left for the designer to define. In our experiments, as proof-of concept, we allow two medium-sized (100K equivalent gates) blocks for each eFPGA.

4.3 The eFPGA model

The eFPGA model basically consists of two parts: structure and functionality. The structure models the internal networking of the FPGA block, its available area and resources. Functionality consists of a standard TLM2 router that forwards transactions to the

appropriate SystemC module or Python function, as shown in Figure 5. The eFPGA has two TLM ports: a configuration port, used to receive bitstreams, and a data port, used by the PEs to access hardware modules. The structure part contains a configuration data structure, that stores information on the currently configured blocks, the references to their implementation, address spaces, etc. The CE manipulates this structure whenever a reconfiguration takes place, and uses the configuration port to transfer bitstream data. The eFPGA features a fixed number of reconfigurable blocks, available in different sizes. For the sake of simplicity we assume three block sizes: small, medium, and large. We assume that a function can be implemented in a block only if its size is smaller than the block's; also, each block has a specific reconfiguration delay, given by a setup time plus the time needed to transfer the bitstream from memory.

The functional part of the eFPGA implements the communication between the data port and the reconfigurable blocks via a TLM2 router. The router checks the destination address of the transaction against a map in the configuration data structure, and it obtains a reference to the hardware module implementing the target function. The simulator elements modeling the reconfigurable hardware have to implement a pre-defined interface, i.e. they have to extend a C++ class or its equivalent Python wrapper in order to be used in our system. This is due to the fact that no SystemC module can be created during simulation, so the eFPGA uses a set of TLM "slots" defined at compile time, that forward their transactions to a C++ or Python functor. Applying the interface to an existing SystemC module simply consists of writing a small wrapper forwarding input parameters to the module's ports. Concerning scripting, the use of ReSP and Python allows for very powerful design space exploration: at any time during simulation it is possible to convert a software function for a hardware model written in Python, also specifying its latency and its size. This means that the designer can quickly explore alternatives by adding and removing functions in the reconfigurable hardware (taking into account area, memory and latency requirements) and observing the effects on the application performance.

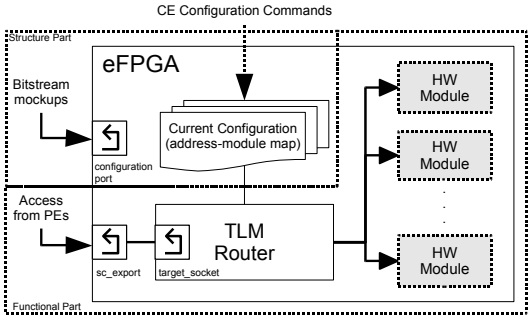

Figure 5. The eFPGA model

5 Experimental Results

The methodology presented in this work has been validated using a popular benchmarking suite for multi-processor systems, namely NASA Parallel Benchmarks (NPB) [11]. Two sets of tests have been performed, with two different purposes:

1. Evaluate the framework for the exploration of different reconfiguration alternatives. To prove this concept, the BT application of the NPB suite was tested and a sub-set of its routines were examined in order to determine the advantages and disadvantages of their implementation using reconfigurable hardware.

2. Evaluate the overhead of using the Python scripting language for specifying the functionalities which have been moved from a software to a hardware implementation. For this, we measured, on a synthetic benchmark, the difference in simulation speed between a C++ and a Python implementation of the same hardware module.

In our tests we used ReSP [4] configured with four cycle-approximate ARM ISS (at 500MHz) with caches, one eFPGA module, two RAM banks, the Configuration Engine and the Reconfiguration Manager. All the components are connected by a 250MHz, 32-bit arbitrated bus. Figure 2 shows a template of the instantiated architecture. Concerning reconfigurable logic, we assume the following:

- There are two possible eFPGA configurations: one module with one reconfigurable block, and one module with two blocks
- The blocks are medium-sized, of roughly 2000 logic cells and the configuration bitstream for each block is about 500Kbit. This number is based on Xilinx FPGAs bitstreams [1]
- Whenever a block is reconfigured, its bitstream is downloaded from RAM via DMA on the shared bus, and the block is ready to be used after transfer plus a setup time of 10μs
- The substitution policy is random

All tests have been performed on a Core 2 Duo Workstation clocked at 2.6GHz.

5.1 NASA Parallel Benchmarks

The NASA Parallel Benchmarks (NPB) [11] suite is composed of parallel programs extracted from a set of important aerophysics applications in *Computational Fluid Dynamics* (CFD). In all the benchmarks the parallelsim is expressed using the OpenMP [2] directives. The BT benchmark, used in this case study, is composed of an implicit algorithm used to solve 3-dimensional compressible Navier-Stokes equations.

Table 1. Profiling of the BT application

Function	ID	Calls #	Time (ms)
`binvcrhs`	BIN	2197	142.65
`matmul_sub`	MUL	2197	82.34
`exact_solution`	EXA	1073	20.29
`matvec_sub`	VEC	2197	16.82
`binvrhs`	RHS	733	14.18
`error_norm`	ERR	2	2.87
...			

Using the proposed methodology, we observe the performance of the system as its functionalities are moved from software to the reconfigurable hardware.

The application was profiled with ReSP, determining the most computationally expensive routines (see Table 1). Those are ideal candidates for a hardware implementation, as they are self contained, they operate on a small set of parameters, and they do not have any side effect. We chose the first four (`BIN`, `MUL`, `VEC`, and `EXA`, all routines for the manipulation of matrices of floating point numbers) for hardware implementation. We assume that the implementation of these functions fits in the available reconfigurable blocks, and that their latency in clock cycles is equal to the number of floating-point operations executed by each of them. Parameters are transferred by the calling processors to the configured blocks via the shared bus. Figure 6 shows the performance trend as these blocks are turned to hardware.

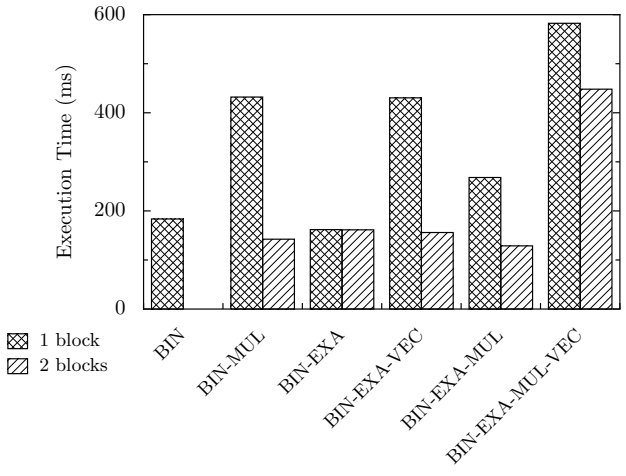

Figure 6. Benchmark execution time as functions are moved to reconfigurable logic, with one and two blocks available

This is due to the reconfiguration and setup time of the eFPGA, and the fact that BIN, MUL and VEC, called alternatively by the four processors, compete for the block ownership. Using a 2-block eFPGA, the situation improves as BIN and MUL (or BIN and VEC) can be called alternatively without delays, but the combination of BIN, MUL, and VEC still incurs long delays. To show the relationship between reconfiguration latency and performance, we ran the benchmark reducing the bitstream size, and with different setup times. Results show that the setup time of the eFPGA has negligible effect: execution time is barely reduced even when time is decreased to zero. Instead, most of the time is spent transferring the bitstream from memory to the eFPGA, reducing the memory bandwidth available for the active processors. The trend is shown in Figure 7: execution time increases with bitstream size.

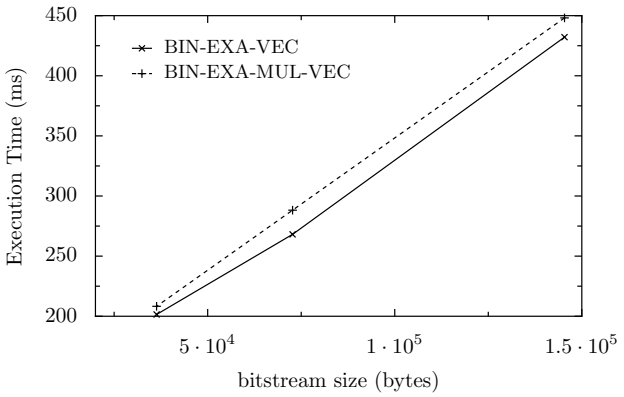

Figure 7. Benchmark execution speed with different bistream sizes for the two configurations most affected by configuration latency

5.2 Python scripting

One of the innovative features of ReSP lies in the integration of SystemC and C++ with the Python programming language. Together with its advantages, Python brings a small penalty in terms of execution speed, being much slower than C++ (as Python is an interpreted language).

To measure the Python's overhead, we simulated a simple program which calls a hardware module a thou-

sand times. In the first half of the runs, the module was implemented in C++, while in the second half it was implemented using Python. Table 2 shows the comparison between the two execution times. Applying the

Table 2. Comparison of the execution time when the hardware models are described with either Python or C++

Language	Avg. Time (ms)	STD (ms)
Python	0.0369	0.005
C++	0.0217	0.0043
Slowdown		59%

same experiment to the previous case study, by defining the functions in Python, for the BIN+EXA+MUL configuration. The measured slowdown is 5%, as the reconfigurable functions are executed 5467 times (see Table 1) over $4.1 \cdot 10^7$ simulated instructions, resulting in $1.3 \cdot 10^{-4} calls/instr$. In comparison, the synthetic benchmark has a function execution rate of $2 \cdot 10^{-2} calls/instr$, over 2 orders of magnitude larger.

6 Concluding Remarks

This paper presented a methodology for the high-level design space exploration of reconfigurable platforms, focusing in particular on devices including embedded FPGAs. Our methodology correctly models bitstream and I/O transfers, setup delays and area size. In addition, the use of transaction-level modeling and reflection allows fast alternative evaluation via scripting using the Python language. Results show how it is possible to quickly explore different architecture alternatives, and how the use of scripting does not excessively affect simulation times. Future work includes more accurate modeling of eFPGA characteristics and constraints, and experiments using blocks with different sizes.

References

[1] FPGA and CPLD solutions from xilinx, inc.

[2] OpenMP application program interface, version 2.5, 2005.

[3] C. Amicucci, F. Ferrandi, M. Santambrogio, and D. Sciuto. SyCERS: a systemC design exploration framework for soC reconfigurable architecture. In *International Conference on Engineering of Reconfigurable Systems & Algorithm*, 2006.

[4] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto. ReSP: A non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration. In *Asia and South Pacific Design Automation Conference 2008 (ASP-DAC 2008)*, 2008.

[5] A. V. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher. Modelling and simulation of dynamic and partially reconfigurable systems using systemC. pages 35–40. IEEE Computer Society, 2007.

[6] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34:171–210, 2002.

[7] A. DeHon and J. Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. pages 610–615, New Orleans, Louisiana, United States, 1999. ACM.

[8] R. Domer. Transaction level modeling of computation, Aug. 2006.

[9] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. pages 327–335, 1989.

[10] P.-A. Hsiung, C.-H. Huang, and C.-F. Liao. Perfecto: A systemc-based performance evaluation framework for dynamically partially reconfigurable systems. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, 2006.

[11] H. Jin, M. Frumkin, and J. Yan. The openMP implementation of NAS parallel benchmarks and its performance.

[12] A. Pelkonen, K. Masselos, and M. Cupak. System-level modeling of dynamically reconfigurable hardware with systemC. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., 2003.

[13] Y. Qu, K. Tiensyrja, and J.-P. Soininen. SystemC-based design methodology for reconfigurable system-on-chip. In *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pages 364–371, 2005.

[14] A. Raabe, P. A. Hartmann, and J. K. Anlauf. Rechannel: Describing and simulating reconfigurable hardware in systemC. *ACM Trans. Des. Autom. Electron. Syst.*, 13:1–18, 2008.

[15] L. Yu, S. Abdi, and D. Gajski. Transaction level platform modeling in systemC for multi-processor designs, 2007.