Integrated Design Environment for Reconfigurable HPC

Lilian Janin, Shoujie Li and Doug Edwards,

School of Computer Science, The University of Manchester,
Manchester M13 9PL, United Kingdom
{lilian.janin, shoujie.li, doug.edwards}@manchester.ac.uk

Abstract. Using FPGAs to accelerate High Performance Computing (HPC) applications is attractive, but has a huge associated cost: the time spent, not for developing efficient FPGA code but for handling interfaces between CPUs and FPGAs. The usual difficulties are the discovery of interface libraries and tools, and the selection of methods to debug and optimize the communications. Our GALS (Globally Asynchronous Locally Synchronous) system design framework, which was originally designed for embedded systems, happens to be outstanding for programming and debugging HPC systems with reconfigurable FPGAs. Its co-simulation capabilities and the automatic re-generation of interfaces allow an incremental design strategy in which the HPC programmer co-designs both software and hardware on the host. It then provides the flexibility to move components from software abstraction to Verilog/VHDL simulator, and eventually to FPGA targets with automatic generation of asynchronous interfaces. The whole design including the generated interfaces is visible in a graphical view with real-time representation of simulation events for debugging purpose.

Keywords: hardware-software interface generator, asynchronous, GALS

1 Introduction

Using FPGAs to accelerate HPC applications involves a huge cost in time. Developing efficient FPGA code is far from being the most time-consuming part of the process. The main problem usually comes from handling the interface between CPU and FPGA: figuring out which libraries to use and how to debug the communications. In fact the difficulties are:

- the choice and description of the interface between the main software and the FPGA implementation;
- how to setup an environment that enables the software programmer to co-design and debug his HPC+FPGA application.

One feature that is usually difficult to achieve is the ability to design and debug the HPC+FPGA application on a separate non-HPC host. This leads to tremendous increases in efficiency as the programmer is free from the HPC constraints: remote text-based terminal, delays for processes to be scheduled, and dynamic compute node

allocation. Of course, the debugging environment also needs, at some point, to be able to target directly the HPC environment as specific bugs may appear at that stage only. The design environment proposed here allows HPC designers to:

- Design and debug their whole design on a non-HPC host, by linking the software code to Verilog or VHDL simulators;
- Remotely target the real HPC system while keeping debugging feedback in the IDE;
- Move components one by one from software abstractions to hardware with the new interfaces being automatically regenerated.

1.1 Background

In order to open up the FPGA market to software programmers, a variety of C-like programming languages have appeared and are available to the HPC programmer [1]: Mitrion-C [2], Celoxica with Handle-C [3], and Nallatech Dime-C [4] being the main ones. These languages provide a higher level of abstraction than conventional Verilog or VHDL and appear more familiar to software HPC developers, leading to shorter development times. However, one of the real benefits of these languages is that they are provided with complete integrated design environments, pre-configured for specific HPC systems. These environments are able to handle the complexity of interfacing CPUs to FPGAs themselves, freeing the user from what we believe is the most difficult task.
The design environment presented in this paper provides similar benefits for interfacing automatically the software running on CPUs to FPGA code described in Verilog or VHDL. Programming in Verilog and VHDL also has some desirable properties: code efficiency and control of the implementation details are sometimes necessary as the FPGA clock runs ten times slower than CPU clocks. Although these HDL languages may require significantly longer development times, their availability for small efficient accelerators is important. With experience, shorter design times can also be achieved as pre-programmed IPs and open-source modules are also available for re-use in these hardware languages.

2 GALAXY Design Framework

The GALAXY framework [5] was originally designed for GALS (Globally Asynchronous Locally Synchronous) embedded system design. It aims at providing an environment of development for iterative design and prototyping of embedded systems where the circuit designer can refine the system description from high levels of abstraction to lower levels, and from software simulation to FPGA prototyping. Components are handled independently at any level of abstraction, targeting any simulator, and the communication interfaces between components, between abstractions and between simulators are automatically regenerated for each simulation. Through the use of various FPGA prototyping boards, we discovered that the GALAXY IDE and tools could help greatly in the task of HPC acceleration.

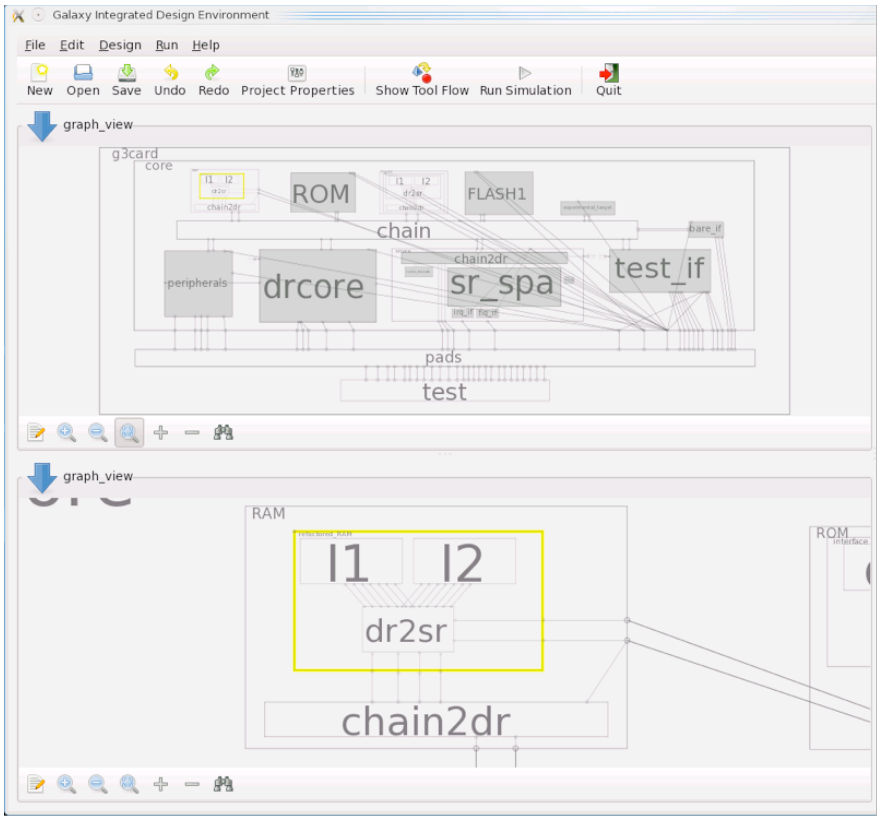

Fig. 1. GALAXY Integrated Design Environment – Dual Graph View.

The framework provides a graphical IDE where software and hardware components are represented as entities (Fig. 1). Each of these components can have multiple implementations (for example an FFT component can have a software implementation in C calling a library and a hardware implementation in Verilog) and implementations can be switched from one to another at the click of a mouse.

For each component the user selects a "simulation target" (where simulation actually includes anything from software execution to FPGA boards) dependent on its source code description: a C description can be executed on host, whereas a Verilog description can be simulated in a Verilog simulator or synthesized and sent to an FPGA. If C synthesizers are available and added to GALAXY's tool flow system, Verilog simulators and FPGA targets become automatically available to components described in C (this is also applicable to any other language).

When two connected components are set to different simulation targets, the communication links are replaced by asynchronous components following a delay-insensitive protocol. This allows the user to experiment with several architectures before optimizing the critical paths, and appears to be an efficient way to proceed.

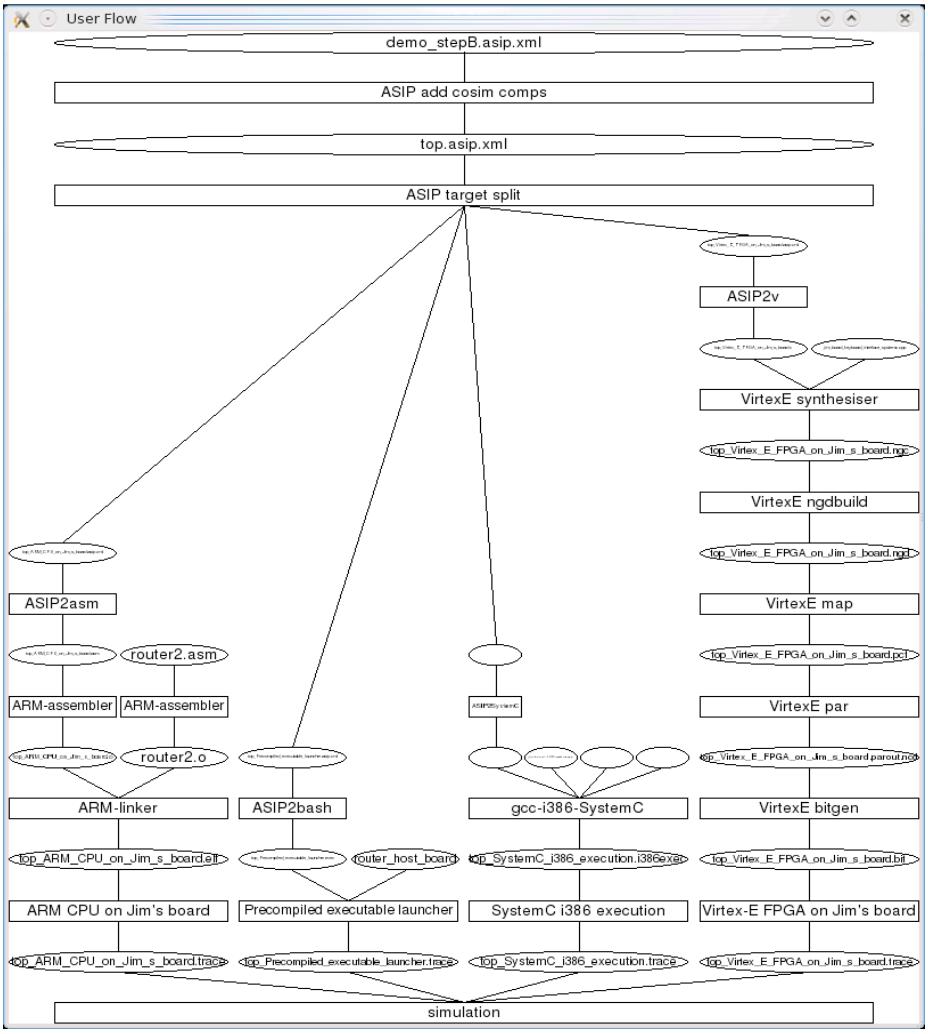

Fig. 2. GALAXY Tool Flow View showing back-end tools.

2.1 The ASIP (Asynchronous-Synchronous IP) XML format

ASIP is a standard component-based description format describing the tree of components making up a circuit. It was developed as part of the GALAXY project. Starting from a top-level component, each component is described in a standard hierarchical way, with a specification of its interfaces, sub-components and connections between sub-components. The leaf components contain a description of their associated source code in any HDL.

What makes ASIP interesting is a set of special constructs, which define the assumptions made to be able to refine a system into asynchronous parts and the constraints on the system description these assumptions impose:

- Asynchronous channels: In the description of the interface and connections, the standard wire and TLM socket types are available, but also an asynchronous channel type. Asynchronous channels are associated to asynchronous protocols. This new channel type allows further design exploration, where the designer can try out various protocols, for example to check for link efficiency. Protocol adapters are automatically added in the GUI where necessary, clearly showing to the designer where bottlenecks may arise.
- Multiple implementations: For each ASIP component, the designer can provide multiple implementations. They must describe the same behaviour and share the same interface, but can be in different languages and at different levels of abstraction.
- Multiple interfaces + transactors: This feature was brought in after long discussions, and makes the GALAXY framework unique: for each ASIP component, the designer can provide multiple interfaces! The problem is that other components in the system might expect one particular interface, and letting the user switch a component's interface could invalidate some connections. For this reason, some constraints apply: proper use of multiple interfaces can be achieved by providing interfaces that are functionally equivalent. The ASIP format encourages this by requiring transactors between the various interfaces. In practice, interfaces are supports for the same communications at different levels of abstraction, with/without debugging signals, and to support synchronous-asynchronous IP wrapping in the GALS context.

An ASIP component can therefore have many implementations and multiple interfaces. Adapters between the different interfaces are included in the ASIP description as transactors, letting the designer select an interface at one level of abstraction for the IP, and a different level of abstraction for its implementation. For example, if an IP is included in a system using its pin-level interface, and the architect needs to simulate it at the SystemC TLM level, an inconsistency is raised and the transactor "pin-level to TLM" is automatically inserted to simulate the component using its TLM description and adapt the transactions to the pin-level interfaces of the connected components. The ability to provide and switch easily between multiple implementations and interfaces, together with the presence of transactors between interfaces, allow a very efficient design space exploration, where IPs can be switched between levels of abstraction in a single operation. In most cases, the transactors can even be automatically generated, for example when TLM sockets are mapped to asynchronous channels.

2.2 GALAXY Back-end Tool Flow

ASIP descriptions are processed by a collection of back-end tools to achieve the regeneration of interfaces for transparent co-simulation for every change in the system architecture. The ASIP flow is as follows (see also an example with four simulation targets in Fig. 2):

- *Asip-add-cosim-comps* inserts co-simulation components in the ASIP file. Each time two connected components are set to be simulated on a different simulator, the connection is split into two and a co-simulation component is added at each end. The co-simulation components behave as if they were wirelessly transferring the data to each other. After this step, the graphs of components corresponding to each simulator are fully disconnected from each other, even though they are still all contained in the same ASIP file. In a HPC environment, these components are programmed to use the HPC communication libraries provided with the FPGA system.
- *Asip-target-split* creates one ASIP file per simulator. All the components targeting a specific simulator are copied to the corresponding file. Incidentally, a flattening of the ASIP structure is also performed at this stage. All the hierarchy is removed and each resulting file is made of one top level components containing directly all the leaf components.
- Each ASIP file is then processed by the code generators *asip2systemc, asip2v, asip2vhdl, asip2asm* or *asip2bash* to generate the top-level source code in an appropriate language for the selected simulator. Keeping the same structure as the input ASIP file, the generated source code is a top-level code instantiating and linking together the various IP source codes. Due to the ASIP flattening occurring in *asip-target-split*, the generated source code is actually a top-level procedure instantiating user-written modules. Most of the time, intermediate modules are created to rename and reorder the signals in order to match the user-defined component interfaces.
- Finally, each generated source code undergoes its own flow as shown in each branch in Fig. 2, specific to the targeted simulator. It can be compiled, synthesized, placed and routed, sent to FPGAs, executed on the host or interpreted. The information about the available tools is stored in the Tool Flow System.

2.3 Tool Flow System: Execution of Tool Flows

The GALAXY IDE also serves as a front-end to launch all the tools, internal back-end tools or external tool flows. A tool flow window allows the user to control these tools, and an execution window provides means of interaction with these tools.
The Tool Flow System is the gateway to link external (vendor or open-source) tools to the GALAXY framework. It has a well-defined interface to facilitate the integration of new tools and tool flows. Its aim is to generate, from the knowledge of available tool flows, an execution sequence of tools to go from the ASIP and IP source codes to the simulators. To achieve this, the tool flow database is made of three main sections:

- available file formats (including how to recognise them, e.g. from their extension);
- available simulators, specifying which input file formats they require;
- available translators (tools able to convert one file format into another, such as compilers and synthesizers), with their required input and produced output file formats.

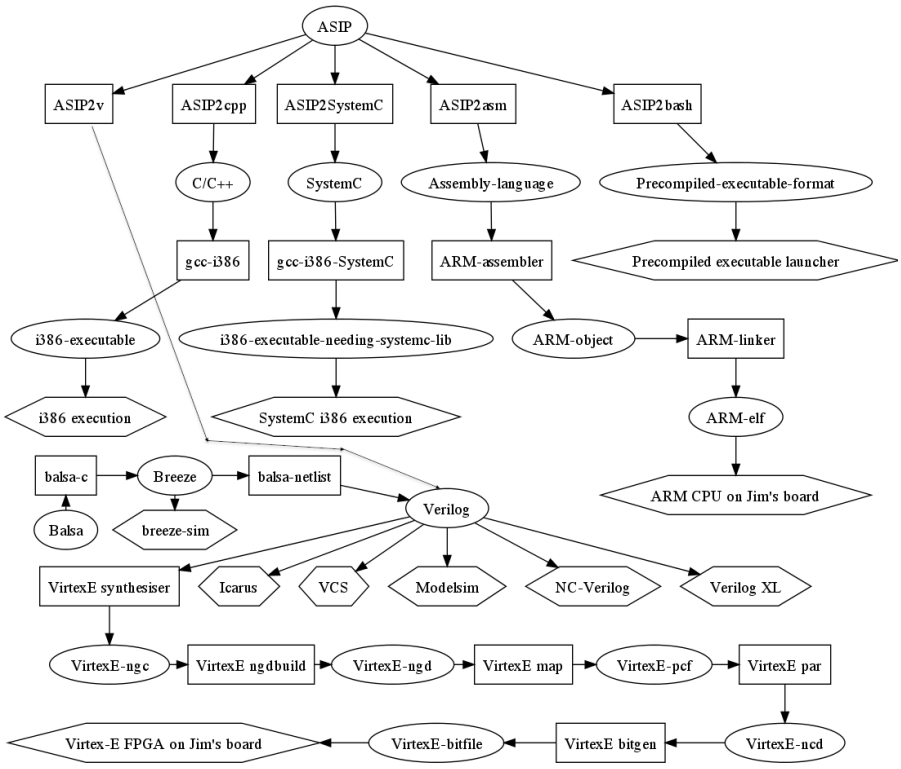

Fig. 3. GALAXY Internal and External Tool Flow.

From this information, a graph of tools and file formats is created (Fig. 3), and appropriate execution sequences are generated when the user desires to "simulate file X with simulator Y".

2.4 Ability to design and debug on a separate non-HPC host

The GALAXY IDE is usually running on a host with graphical display, therefore not directly on the HPC system. In the first stage of the design process, the HPC designer can design and simulate the HPC application on the non-HPC host, giving him the advantage of faster access and direct control over the hardware simulators. This leads to tremendous increases in efficiency, as the programmer is free from the HPC constraints: remote text-based terminal, delays for processes to be scheduled, dynamic compute node allocation.

A good way to run and debug the system on a single non-HPC host is to have the FPGA code simulated in a Verilog simulator, or even better: a graphical debugger like ISE.

However, parallel HPC code cannot always be compiled and executed on any host, due to the links to MPI libraries, but a non-MPI test harness is usually an acceptable way to start the design of the FPGA code. Of course, the debugging environment also

needs, at some point, to be able to target directly the HPC environment, as specific bugs may appear at that stage only. This is achieved by the Tool Flow System after configuration of scripts to handle the transfers between the host and the HPC system. Scripts can include the ability to access HPC systems behind gateways and submit jobs in queues.

3. CONCLUSIONS AND FURTHER WORK

The GALAXY framework, originally designed for GALS embedded system design, happens to be mature for the design of accelerated HPC applications. It allows HPC programmers to apply an iterative strategy for the use of HPC FPGA boards. They can design and debug their whole design on a non-HPC host, by linking the software code to Verilog or VHDL simulators, and then remotely target the real HPC system while keeping debugging feedback in the IDE. Components can be moved one by one from software abstractions to hardware with the new interfaces being automatically regenerated.

The generated asynchronous interfaces happen to be efficient enough for a first version of the user's HPC application. They allow the user to experiment with several architectures before optimizing the critical paths, and appear to be an efficient way to proceed.

Although no benchmark is available yet, the GALAXY framework presented in this paper has been augmented for HPC by using a Cray XD1 with Xilinx Virtex 4 FPGAs. The Xilinx tool flow has been integrated in the tools, and we are now working on a demonstrator.

Acknowledgements

This project is funded by the European Seventh Framework Programme (FP7).

We would like to express our thanks to the CCLRC Daresbury Laboratory and their great team for providing access to their Cray XD1.

References

1. R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin and C. Kitchen , "An overview of FPGAs and FPGA programming; Initial experiences at Daresbury", Computational Science and Engineering Department, CCLRC Daresbury Laboratory, 2006.
2. S. Mohl, "The Mitrion-C programming language," Mitrionics Inc., 2006. [online]. Available: http://www.mitrionics.com/
3. Celoxica Ltd. http://www.celoxica.com
4. Nallatech Dime-C. http://www.nallatch.com
5. http://www.galaxy-project.org